Chapter 5
Measuring dialect distances

In the Chapters 3 and 4 we described how distances between phonetic segments are found. When the segments are aligned, we are able to find distances between words, and in turn between language varieties. The way in which distances are found between words and between language varieties is the topic of this chapter. The central algorithm in this chapter and in this research is the Levenshtein distance, a method that allows distances between words to be measured. This algorithm may be applied to both transcriptions of words and to the representations of the acoustic signals of word samples. The application of the Levenshtein distance to transcriptions of words is described in Section 5.1. This approach uses the phonetic segment distances as measured in the Chapters 3 and 4. In Section 5.2 we describe the application of the Levenshtein distance to acoustic word samples. In this approach a transcription is only used for finding the number of phonetic segments per word. The segment distances as measured in the Chapters 3 and 4 are not used.

5.1 Levenshtein distance using transcriptions

5.1.1 Sequence comparison

Sequence comparison is used in many different fields. Kruskal (1999) gives an overview. First, Kruskal mentions the application to molecular biology, where sequence comparison is used for the comparison of macromolecules. An example that is more related to our research is the application of sequence comparison to speech and speaker recognition. Sequence comparison is also used for correction of typing errors on a computer or keypunch machine, for the comparison of comparable computer files and for error control of codes which are transferred by e.g. radio or telegraph. Levenshtein (1966) ‘presented the earliest known use of a distance function that is appropriate in the presence of insertion and deletion errors’ (Kruskal, 1999, p. 5). Sequence comparison is also applied in gas chromatography,
a physical method used to separate and/or analyse complex mixtures. A ‘mixture is swept by a continuous stream of nonreactive carrier gas through a long, densely packed column of special material’ (p. 6). Components with strong attraction to one part of the column move more slowly than those with weak attraction. ‘The components emerge at different times over a period of minutes or hours’ (p 6). A chromatogram shows different peaks in time. The peaks correspond to the intensities of the different components in the sample mixture. Chromatograms are sequences that are compared to each other. Also related to our research is the application of sequence comparison to bird song. In ‘some bird species, song is an important means of communication, which is learned by the young from their elders, and it has dialect-like variation from place to place’ (p. 7). Another application of sequence comparison is found in the comparison of stratigraphic sequences, tree rings and varves (‘annual layers of sediment, generally clay, in which it is possible to count the years’, p. 7). More related to our research, sequence application is applied to collation of different versions of the same text. Furthermore, sequence comparison is found when ‘computer processing handwritten material such as signatures and line drawings’ (p. 8). Comparison of “brain waves” in response to a stimulus may also be application of sequence comparison.

In our research we apply sequence comparison to the comparison of different pronunciations corresponding to different language varieties in order to measure the distance between them. Kruskal mentions several methods which require that sequences have the same length. Examples are Hamming distance (the number of positions in which the corresponding elements are different), Manhattan (or city-block) distance, and Euclidean distance (see Section 3.6.2.5). However different pronunciations will not have the same length in many cases. Also, the correspondences made in the methods just mentioned may not always be correct. E.g. afternoon may be pronounced as [æftənən] in the dialect of Savannah, Georgia and as [æftərnən] in the dialect of Lancaster, Pennsylvania.³ Assume we compare both pronunciations using the Hamming distance. When ignoring diacritics this is done as follows:

\[
\begin{array}{cccccc}
\text{æ} & \text{ə} & \text{ft} & \text{ɔ} & \text{n} & \text{ən} \\
\text{æ} & \text{f} & \text{t} & \text{ə} & \text{rn} & \text{un}
\end{array}
\]

We get a cost of 5. However, we see that elements which correspond to one another, are unfortunately not regarded as corresponding elements when calculating the Hamming distance. The consequence is that the distance calculated between the two pronunciations is too high.

Both the length and the correspondence problem are solved when using the Levenshtein distance. This algorithm is able to deal with different lengths calcu-

³The data is taken from the Linguistic Atlas of the Middle and South Atlantic States (LAMSAS) and available via http://hyde.park.uga.edu/lamsas/.
5.1. LEVENSHTEIN DISTANCE USING TRANSCRIPTIONS

lating the distance on the basis of probable correspondences. Although Leven-
shtein applied his algorithm to error control of codes which are transferred by e.g., radio or telegraph, the algorithm may be applied to all cases mentioned at the beginning of this section. For the comparison of genetic macromolecules and the recognition of human speech the algorithm has actually been used. In speech recognition the algorithm is often referred to as (dynamic) time-warping or dynamic programming. The Levenshtein distance was first applied by Kessler (1995) to dialect comparison. He used the algorithm for the comparison of Irish Gaelic varieties. In our research we have used the algorithm for the comparison of Dutch and Norwegian varieties. See Bolognesi and Heeringa (2002) for an application to Sardinian dialects.

5.1.2 Minimum cost

Fundamental to the idea of the Levenshtein distance is the notion of string-changing operations. To determine the extent to which two strings differ from each other, an inventory of what operations can change one string into another should be made. The operations available are:

- **Deletions**
  Delete an element from the string.

- **Substitutions**
  Replace an element from the one string by an element of the other string.

- **Insertions**
  Add an element to the string.

Insertions and deletions are also referred to as *indels*. To each of the operations weights are assigned. In the simplest case all operations have the same weight, e.g. 1 (Gusfield, 1999, p. 218). When applying these weights to the comparison of word pronunciations, we judge that roughly speaking, substitutions will be about equally noticeable as indels from the perceptual point of view. We illustrate the use of the operations with an example. As mentioned in Section 5.1.1 *afternoon* may be pronounced as \([æfˈtəɹn]n\) in the dialect of Savannah and as \([ˈæftəɹn]n\) in the dialect of Lancaster. Ignoring diacritics for this moment the one pronunciation can be changed into the other as follows:
However, the procedure which is followed here is round about. It can be done much more efficiently in this way:

\[
\begin{array}{ccc}
\text{æəftərum} & \text{delete } ə & 1 \\
\text{æəftərum} & \text{delete } f & 1 \\
\text{æəətərum} & \text{subst. } t/r & 1 \\
\text{æəənəm} & \text{insert } f & 1 \\
\text{æəənəm} & \text{insert } t & 1 \\
\text{æəftənəm} & \text{subst. } u/u & 1 \\
\text{æəftənəm} & & \\
\end{array}
\]

Both examples illustrate that is is possible to change one pronunciation to the other in many ways, often resulting in different costs. We are interested in the set of operations with the least cost that change a pronunciation \(w_1\) into a pronunciation \(w_2\). This is equal to the Levenshtein distance \(d(w_1, w_2)\). Given that there are many different sets of operations mapping \(w_1\) to \(w_2\), it is not obvious how to determine the minimal set, and it is even less obvious how to determine it efficiently. The Levenshtein algorithm, however, accomplishes both these tasks.

### 5.1.3 Operation weights

Pronunciations are compared on the basis of their segments. When using the phone representation (see Section 3.1.1) the cost of substitutions and indels is set to 1, just as in the examples in Section 5.1.2. When using feature representations (see Sections 3.1.2, 3.1.3 and 3.1.4) or acoustic representations (see Sections 4.3.1, 4.3.2 and 4.3.3) the weights gradually vary. For substitutions, the weight is equal to the distance between the corresponding segments calculated according to the chosen segment representation. For indels the weight is equal to the distance between the segment to be inserted or deleted and ‘silence’. This weight also depends on the segment representation chosen and is defined for each of the feature representations and acoustic representations separately. Gradual substitutions and indels may be based on both linear and logarithmic segment distances (see Section 3.7 and Section 4.7).
5.1. LEVENSHTEIN DISTANCE USING TRANSCRIPTIONS

5.1.4 Allowed matches

To accord with syllabification in words, the Levenshtein distance should be based on an alignment with plausible matches. In our implementation of the algorithm the basic rule is that a vowel may normally only match with a vowel and a consonant normally only with a consonant. However, the [w] and the [j] may also match with vowels, and the [u] and the [i] may also match with consonants.

For some representations, vowels can be compared to consonants, for other representations this is impossible. Checking the different representations we see the following:

- Using the phone representation, sounds have no real definitions. If they are equal, the distance is 0, otherwise 1. The comparison of a vowel with a consonant is possible in principle (see Section 3.1.1).

- In the feature system of Hoppenbrouwers & Hoppenbrouwers (H & H) all features apply for both vowels and consonants, which basically offers the possibility of comparing a vowel with a consonant (see Section 3.1.2).

- In the feature systems of V & C and A & B it is not possible to compare a vowel with a consonant. Since we defined the [i], [u], [j] and [w] as both vowel and consonant, these sounds are exceptions (see Sections 3.1.3 and 3.1.4).

- When using an acoustic representation the comparison of a vowel with a consonant can be easily made (see Chapter 4).

For those representations where it is possible to compare a vowel with a consonant, we will not allow all vowel-consonant matches. And it is indeed not likely that a [p] will change into an [a]. On the other hand, it is not unusual that e.g. an [r] matches with an [o]. For example two possible pronunciations for the Dutch word \( \textit{vier} \) ‘four’ are [fiːr] and [fiːo]. Here we want the ending [r] and the ending [o] to match with each other. Therefore, we allow the match of a schwa with a sonorant.

5.1.5 No swap operation

A phenomenon which can be found in dialect data is metathesis. For example the equivalent of ‘wasp’ is pronounced as [vɛ̃sp] in Standard Dutch and also in the dialect of Amsterdam, and as [uɛ̃ps] in the dialects of Utrecht and Den Haag.\(^2\) Using only substitutions and indels the minimum cost is found with the following alignments:

\(^2\)The pronunciations are taken from a data set compiled by Renée van Bezooijen, University of Nijmegen, in 2000.
Actually the s and the p in the first transcription should correspond with the s and the p in the second transcription. In that way no segment distances are found. When using the phone representation it seems reasonable to assign a weight of 1 to the swap operation, the same weight as assigned to substitutions and indels. However when using gradual weights (see Section 5.1.3), the swap operation should also be weighted gradually. Some segments may be easily swapped, for example a plosive and a non-plosive (as in our example), or a vowel and a consonant, but for other segments this may be (nearly) impossible. Once these gradual weights are found, they should be scaled so that they are in the right proportion to the weights of substitutions and indels. We have not yet succeeded in finding gradual and correctly scaled weights for the swap operations. This is an interesting topic for future work.

5.1.6 Calculation of distance

In this section we explain the calculation of the Levenshtein distance on the basis of both pronunciations of *afternoon*. We call [æftənun] $S_1$ and [æftənun] $S_2$. The number of segments in $S_1$ is $m$ and in $S_2$ $n$. We see that $m = 8$ and $n = 8$. The Levenshtein distance calculates the minimum cost needed to change $S_1$ into $S_2$. For this we use a matrix $\text{dist}$ of size $(m + 1, n + 1)$. The rows are numbered from 0...$m$ and the columns from 0...$n$. The cell $\text{dist}[0,0]$ gets the value 0. We traverse the matrix $\text{dist}$ row by row, assigning values to the other cells. We begin with row 0, and within each row, we always begin with column 0 (only in the zeroth row do we start with the first column). We call the current row number $i$ and the current column number $j$. For each cell in the matrix, we always have to look at three possibilities (to obtain a minimum):

1. **Deletion** of the i-th segment from $S_1$. We determine $weight(S_1,\emptyset)$. We take the sum of this weight and the value in the cell above the current one: $\text{dist}[i - 1, j]$. This sum is assigned to the temporary variable $\text{upper}$. This operation is only considered when $i > 0$.

2. **Substitution** of the i-th segment of $S_1$ by the j-th segment of $S_2$. We look up $weight(S_1,S_2)$. We take the sum of this weight and the value in the cell above and to the left of the current one: $\text{dist}[i - 1, j - 1]$. This sum is kept in the temporary variable $\text{upperleft}$. This operation is only taken into account when $i > 0$ and $j > 0$. 
5.1. LEVENSHTEIN DISTANCE USING TRANSCRIPTIONS

function Levenshtein_distance(S1,S2)
begin
   for i:=0 to m do begin
      for j:=0 to n do begin
         upper=upperleft=left:=maxint;
         if i>0
            then upper:=dist[i-1,j]+weight(S1[i],ø);
         if i>0 and j>0
            then upperleft:=dist[i-1,j-1]+weight(S1[i],S2[j]);
         if j>0
            then left:=dist[i,j-1]+weight(ø,S2[j]);
         dist[i,j]:=min(upper,upperleft,left);
         if dist[i,j]=maxint then dist[i,j]:=0;
      end
   end
   Levenshtein_distance:=dist[m,n];
end

Figure 5.1: Levenshtein algorithm in pseudo-code. The algorithm works dynamically, so that, for each \( p_1, p_2 \) prefix pair of S1, S2, it determines the least cost of operations mapping \( p_1 \) to \( p_2 \). The number of segments in S1 is \( m \) and in S2 \( n \).

3. Insertion of the \( j \)-th segment in S2. We compute \( \text{weight}(\varnothing, S_2_j) \). We take the sum of this weight and the value in the cell left of the current one: \( \text{dist}[i, j - 1] \). The sum is retained in the temporary variable \( \text{left} \). This operation is only considered when \( j > 0 \).

Now, we take the minimum of the three values, \( \text{upper} \), \( \text{upperleft} \) and \( \text{left} \), and the current cell takes it as value:

\[
\text{dist}[i,j] \leftarrow \min(\text{upper},\text{upperleft},\text{left})
\]

In this way we ensure that paths arise only by adding minimally to minimal-cost cells. This guarantees that the least distance is computed. Once we have traversed the entire matrix, and computed values for all cells, then the distance – the least cost of operations mapping from S1 to S2 – is found in the cell.
dist\([m,n]\). This is the Levenshtein distance between the strings. The algorithm in pseudo-code is shown in Figure 5.1.

The matrix below shows the application of the procedure to our example. Initially dist\([0,0]\) gets the value 0. In most other cells four values are given. The variables upper, upperleft, left are given respectively in the upper right, upper left and lower left of a cell. The minimum of these three variables is given in the lower right of a cell. Note that in the 0-th row only the variable left (insertions) could be calculated and in the 0-th column only the variable upper (deletions) could be calculated. The final distance between the two pronunciations is the lower right value in cell dist\([8,8]\): 3. In Section 5.1.7 we explain how the cheapest path from dist\([8,8]\) to dist\([0,0]\) can be recovered.

<table>
<thead>
<tr>
<th></th>
<th>(\emptyset)</th>
<th>(æ)</th>
<th>(ə)</th>
<th>(f)</th>
<th>(t)</th>
<th>(ə)</th>
<th>(n)</th>
<th>(u)</th>
<th>(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\emptyset)</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>æ</td>
<td>1 0 2 3 4 5 6 7 8</td>
<td>1 2 0 1 1 2 2 3 4 5 6 7 8 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>2 3 1 1 2 1 2 1 1 2 2 3 4 4 5 5 6 7 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>3 4 2 3 2 3 2 3 2 3 1 2 2 3 3 4 4 5 5 6 7 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ə</td>
<td>4 4 3 2 3 3 3 3 3 3 1 3 4 4 5 5 5 5 6 7 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>5 5 4 4 3 4 3 4 3 4 3 3 2 3 3 4 4 5 5 6 7 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>6 6 5 5 4 4 4 4 4 4 4 3 2 3 3 4 3 3 3 3 3 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u</td>
<td>7 7 6 6 5 5 5 5 5 4 4 3 3 4 4 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>8 8 7 7 6 6 6 6 6 5 5 4 4 4 4 3 3 3 3 3 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.1.7 Tracing backwards

Once the distance between S1 and S2 is computed, it is possible to find the corresponding alignment(s) which show the mapping of S1 to S2. For this purpose it is easy to set pointers when traversing the matrix for the first time. For each cell this is done as follows:

1. If the variable upper is equal to the minimum value, set a pointer from the current cell to the cell above.
2. If the variable upperleft is equal to the minimum value, set a pointer from the current cell to the cell leftabove.

3. If the variable left is equal to the minimum value, set a pointer from the current cell to the cell left.

When \( k \) variables are equal to the minimum, there are at least \( k \) paths from \( \text{dist}[0,0] \) to the current cell which results in the minimum cost for that (sub)sequence. The matrix below shows the pointers for our example.

<table>
<thead>
<tr>
<th></th>
<th>( \emptyset )</th>
<th>( \text{æ} )</th>
<th>( \text{ø} )</th>
<th>( \text{f} )</th>
<th>( \text{t} )</th>
<th>( \text{ø} )</th>
<th>( \text{n} )</th>
<th>( \text{u} )</th>
<th>( \text{n} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \emptyset )</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{æ} )</td>
<td>↑</td>
<td>( \emptyset )</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{f} )</td>
<td>↑</td>
<td>↑</td>
<td>( \emptyset )</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{t} )</td>
<td>↑</td>
<td>↑</td>
<td>( \emptyset )</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{ø} )</td>
<td>↑</td>
<td>↑</td>
<td>( \emptyset )</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{r} )</td>
<td>↑</td>
<td>↑</td>
<td>( \emptyset )</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{n} )</td>
<td>↑</td>
<td>↑</td>
<td>( \emptyset )</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{u} )</td>
<td>↑</td>
<td>↑</td>
<td>( \emptyset )</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{n} )</td>
<td>↑</td>
<td>↑</td>
<td>( \emptyset )</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The optimum alignment is found by tracing backwards. We start at \( \text{dist}[m,n] \) and follow along the arrows to obtain a path all the way to \( \text{dist}[0,0] \). The alignment is read off from the path in reverse order. The arrows have the following meaning:

1. A vertical arrow in row \( i \) means: delete \( S_1 \) and place \( [S_1 \emptyset] \) in the alignment.

2. A diagonal arrow in row \( i \) and column \( j \) means: substitute \( S_1 \) by \( S_2 \) and place \( [S_1 S_2] \) in the alignment.

3. A horizontal arrow in column \( j \) means: insert \( S_2 \) and place \( [\emptyset S_2] \) in the alignment.

When \( k \) arrows are found in a cell, there are at least \( k \) paths from \( \text{dist}[0,0] \) to the current cell. In our example there is only one path which gives the minimal cost. The shaded cells make up this path. This path corresponds with the following alignment:
5.1.8 Normalization of length

When computing the distance between two sequences, in general the distance between longer sequences will be greater than the distance between shorter sequences. The longer the sequences, the greater the chance of differences between them. If we used these distances directly, then longer words would contribute disproportionally to the estimation of distances between varieties, which does not accord with the idea that words are linguistic units. Therefore, we normalize the distance by a factor that is related to the length of the sequences. Assume four different string pairs are aligned in the following way:

\[
\begin{array}{ccc}
  a_1 & a_2 & a_3 \\
  b_1 & b_2 & b_3 \\
  1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{ccc}
  æ & ø & f & t & æ & ø \\
  æ & ø & f & t & æ & ø \\
  1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{ccc}
  a_1 & a_2 & \emptyset \\
  b_1 & b_2 & b_3 \\
  1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{ccc}
  a_1 & a_2 & \emptyset \\
  \emptyset & \emptyset & b_3 \\
  1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{ccc}
  a_1 & a_2 & \emptyset \\
  \emptyset & b_2 & b_3 \\
  1 & 1 & 1 \\
\end{array}
\]

Since substitutions and indels have the same weight for each of the string pairs we get a cost of 3 for all string pairs. To do justice fully to the fact that all sequence operations have the same weight, in our opinion the costs of the different string pairs should also be the same after normalizing over the length of the sequences. For (1) the distance can simply be divided by the length of either of the two sequences, so the normalization factor becomes 3. This factor should also be used for the other string pairs. In (2) this factor is equal to the length of the longer sequence. In (3) this factor is equal to the sum of the lengths of both sequences. In (4) the normalization factor can be found less easily. The length of both sequences is equal to 2 (there is no longer sequence since the two sequences have the same length) and the sum of the sequences is equal to 4. Taking the mean of these two values we find the normalization factor: \((2 + 4)/2 = 3\). The different string pairs show that the way in which the normalization factor is found on the basis of the lengths of the sequences is not always the same. However, when dividing by the length of the alignment, for all cases the same normalization factor is found, namely 3.

As described in Section 5.1.7 the minimum cost of changing one sequence into another may correspond with more than one path in the matrix. As a hypothetical example we consider the subsequence [æft] of S1 and the subsequence [æǝf] of S2. The value lowerright in cell dist[3,3] (see Section 5.1.6) gives the minimum cost that is needed to change one subsequence into the other: 2. Examining the
1. bee
   German: \( b i n \bar{o} \)
   Dutch: \( b \bar{e}i \)
   \[ \begin{array}{c|c|c}
   \text{Distance} & 1 & 1 \\
   \hline
   \end{array} \]

2. rabbit
   German: \( k a n i n \bar{c} \bar{o} n \)
   Dutch: \( k \bar{o}n e i n \)
   \[ \begin{array}{c|c|c|c|c}
   \text{Distance} & 1 & 1 & 1 & 1 \\
   \hline
   \end{array} \]

3. kanari
   English: \( k \bar{o}n \bar{e} \bar{r} \bar{i} \)
   Frisian: \( k \bar{o}n a r j \bar{o} \)
   \[ \begin{array}{c|c|c|c|c}
   \text{Distance} & 1 & 1 & 1 & 1 \\
   \hline
   \end{array} \]

Figure 5.2: Three word pairs with two alignments each. The longer alignment on the right is judged as the better one. Diacritics are not taken into account.

The pointer matrix (see Section 5.1.7) it appears that in cell \( \text{dist}_{3,3} \) two pointers are given, one pointing to the cell above and left, and one pointing to the cell above. The result is that there are two possible paths corresponding with the following alignments:

\[
\begin{array}{cc}
ae f t & ae \emptyset f t \\
\bar{a} \bar{e} \bar{a} f & \bar{a} \bar{e} \emptyset f \emptyset \\
1 & 1 & 1 & 1 \\
\end{array}
\]

We judge the alignment to the right as the better one since in this alignment the two \( f \)'s appear as corresponding segments. We get the impression that the longest alignment has always the greatest number of matches. Shorter and longer alignments for more pairs of different pronunciations are given in Figure 5.2.\(^3\) Both alignments give the minimum cost. The examples confirm our conjecture. In the longer alignments more matches are found than in the shorter ones. However, is an alignment with a greater number of matches always better than an alignment with a smaller number of matches? To answer this question, consider that distances should approach human perception as close as possible. Therefore, an alignment should reflect the way in which people perceive differences between pronunciations rather than reflecting the way in which one pronunciation changed into the other in history. From this point of view the longer alignments in the examples 1, 2 and 3 are the better ones. We suppose that in perception people will try to match the common sounds in two different pronunciations, so we prefer the longer alignments.

We normalize by dividing the distance by the length of the longer alignment. This gives the average of the weights used. In our hypothetical example in which

\(^3\)The pronunciations are taken from a data set compiled by Renée van Bezooijen, University of Nijmegen, in 2000.
two pronunciations of the word *aft* are compared, the distance is equal to 2 and the length of the longer alignment is equal to 4. The total cost of 2 is now divided by the length of 4 which gives an average weight of 0.5. When the weights represent percentages (as for the acoustic distances, see Section 4.7), dividing the distance by the length gives the average weight as a percentage. In that case the word distance is expressed as a percentage. In our example, the weights 0 and 1 may be replaced by 0% and 100%. This results in a word distance of 50%. In our example in which two pronunciations of the word *afternoon* are compared the distance is equal to 3 and the length of the alignment is equal to 9. The word distance expressed as a percentage is equal to \( \frac{3 \times 100\%}{9} = 33\% \).

### 5.1.9 Calculation of length

The normalization length is taken to be equal to the length of the alignment. In the previous section we showed that different alignments corresponding with different paths in the matrix \( \text{dist} \) may give the same minimum cost. The different alignments or paths may have different lengths. We explained that we prefer to divide the minimum cost by the length of the longer alignment or path. The way in which the maximum length is calculated is comparable to the way in which the minimum distance is found. The calculation of distance and length is done in the same software module. We use a matrix \( \text{length} \) with the same size as \( \text{dist} \): \((m+1, n+1)\). The rows are numbered from 0...\( m \) and the columns from 0...\( n \). The matrix \( \text{length} \) is traversed the same way as and simultaneously with the matrix \( \text{dist} \). For each cell in the matrix, we regard three possibilities:

1. If \( \text{upper} \) is equal to the minimum cost of the (sub)sequence we assign the value of \( \text{length}[i-1,j] \) increased by 1 to a temporary variable \( \text{Upper} \). Otherwise \( \text{Upper} \) becomes negative. This operation is only used when \( i > 0 \).

2. If \( \text{upperleft} \) is equal to the minimum cost of the (sub)sequence we assign the value of \( \text{length}[i-1,j-1] \) increased by 1 to a temporary variable \( \text{UpperLeft} \). Otherwise \( \text{UpperLeft} \) becomes negative. This operation is only possible when \( i > 0 \) and \( j > 0 \).

3. If \( \text{left} \) is equal to the minimum cost of the (sub)sequence we assign the value of \( \text{length}[i,j-1] \) increased by 1 to a temporary variable \( \text{Left} \). Otherwise \( \text{Left} \) becomes negative. This operation is only possible when \( j > 0 \).

Increasing the value of a previous cell by 1 represents one step in the path. In contrast to the procedure in the distance calculation we take the maximum of the three values, \( \text{Upper} \), \( \text{UpperLeft} \) and \( \text{Left} \), and the current cell takes it as value:

\[
\text{length}[i,j] \leftarrow \text{maximum}(\text{Upper}, \text{UpperLeft}, \text{Left})
\]
Once we have traversed the entire matrix, and computed values for all cells, then the length of the longest alignment which gives the minimum cost is found in \( \text{length}[m, n] \).

The matrix below shows the application of the procedure to our example. In the matrix, the cells of the path which gives the minimum cost are shaded. Initially \( \text{length} \) gets the value 0. For each cell in the matrix the variables \( \text{Upper} \) (upper right in the cell), \( \text{UpperLeft} \) (upper left) and \( \text{Left} \) (lower left) are given. However, they are only given when their corresponding counterparts (\( \text{upper} \), \( \text{upperleft} \) and \( \text{left} \)) are equal to the minimum cost (compare also the pointer matrix in Section 5.1.7). Otherwise a negative value is assigned which is not given in the matrix. The maximum of these three variables is given in the lower right of a cell. The final length is the lower right value of cell \( \text{length}[8, 8] \): 9.

\[
\begin{array}{cccccccccc}
\emptyset & æ & ø & f & t & ø & n & u & n \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
\emptyset & 0 & 0 & 1 & 1 & 2 & 2 & 3 & 3 & 4 & 4 & 5 & 5 & 6 & 6 & 7 & 7 & 8 & 8 \\
æ & 1 & 1 & 1 & 1 & 2 & 2 & 3 & 3 & 4 & 4 & 5 & 5 & 6 & 6 & 7 & 7 & 8 & 8 \\
f & 2 & 2 & 2 & 2 & 3 & 3 & 4 & 4 & 5 & 5 & 6 & 6 & 7 & 7 & 8 & 8 \\
t & 3 & 3 & 3 & 3 & 3 & 3 & 4 & 4 & 5 & 5 & 6 & 6 & 7 & 7 & 8 & 8 \\
ø & 4 & 4 & 4 & 4 & 4 & 4 & 5 & 5 & 5 & 5 & 5 & 6 & 6 & 7 & 7 & 8 & 8 \\
r & 5 & 5 & 5 & 5 & 5 & 5 & 5 & 6 & 6 & 6 & 6 & 6 & 7 & 7 & 9 & 9 \\
n & 6 & 6 & 6 & 6 & 6 & 6 & 6 & 7 & 7 & 7 & 7 & 7 & 8 & 8 \\
u & 7 & 7 & 7 & 7 & 7 & 7 & 7 & 8 & 8 & 8 & 8 & 8 & 8 & 9 & 9 \\
n & 8 & 8 & 8 & 8 & 8 & 8 & 9 & 9 & 9 & 9 & 9 & 9 & 9 & 9 & 9 & 9
\end{array}
\]

5.1.10 Dialect distances

Once we are able to calculate the distance between two pronunciations of a word, we can also find the distance between two varieties. When for \( k \) words the pronunciations are given for both varieties, we get \( k \) word pairs. For each pair we calculate the Levenshtein distance. In this way we get \( k \) Levenshtein distances. Now the distance between the two varieties is equal to the average of the \( k \) Levenshtein distances. When the word distances represent percentages, the distance
between both varieties represents a percentage as well. Because in our research varieties are mostly dialects, the distance between two varieties is referred to as dialect distance. When having \( n \) dialects, for each possible pair of dialects the average Levenshtein distance can be calculated. The corresponding distances can be arranged in a \( n \times n \) matrix.

### 5.1.10.1 Missing transcriptions

When comparing two varieties on the basis of \( k \) word pairs, it may happen for one or more of the pairs, for one or both varieties, that no pronunciation is given. Since we work with average distances, we simply discount the effect of missing transcriptions. This has the same effect as estimating the word distance to be average word distance. The use of average distance has the advantage that it allows us to examine distances between \( n \) dialects for each of the \( k \) words individually, even if for some varieties pronunciations are missing.

### 5.1.10.2 Multiple transcriptions

Sometimes several transcriptions are given for the same word in one variety. Assume in variety 1 the equivalent for ‘house’ is pronounced as [hys] and [hus]. In variety 2 the same two transcriptions are found. The two varieties are equal in the sense that both [hys] and [hus] are possible. When simply calculating the mean of all possible pairs ([hys] vs. [hys], [hys] vs. [hus], [hus] vs. [hys] and [hus] vs. [hys]) we incorrectly get a word distance which is higher than 0. A better approach would be to calculate the mean of all plausible pairs, i.e. pairs with elements that probably correspond to each other. In that case we get the mean distance of the pairs [hys] vs. [hys] and [hus] vs. [hus] which is equal to 0. In this section we propose a procedure that is based on the idea that the mean word distance should be based on the most natural word pairs.

Our current implementation is able to deal with at most ten different pronunciations per word per variety. We illustrate the way in which we process them by an example. Assume word W1 in dialect 1 has the following transcriptions:

\[
a \ b \ c
\]

and word W2 in dialect 2 has the following transcriptions:

\[
x \ y
\]
Dialect 1 has three transcriptions, and dialect 2 has two transcriptions. We duplicate each of the three transcriptions in dialect 1 two times, and each of the two transcriptions in dialect 2 three times. For dialect 1 we get:

\[ a \ a \ b \ b \ c \ c \]

and for dialect 2 we get:

\[ x \ x \ x \ y \ y \ y \]

We see that the number of variants of W1 and W2 is the same, namely 6. We want to find the 6 most likely pairs of variants. This is done by a heuristics. For finding 6 pairs we perform 6 iterations. Within each iteration we find the pair of variants (one of dialect 1 and one of dialect 2) with the the smallest distance. The members of the pair may not already be used in previous formed pairs. The final distance between W1 and W2 is the sum of the word distances corresponding with the 6 pairs divided by 6.

The procedure may be described in more general terms. Assume for word W1 \( m \) transcriptions are given and for word W2 \( n \) transcriptions. Each of the \( m \) transcriptions is duplicated \( n \) times, and each of the \( n \) transcriptions is duplicated \( m \) times. In this way for both W1 and W2 we get \( m \times n \) variants (which are not all unique). The variants of W1 are indexed as \( 1 \leq p \leq m \times n \), and the variants of W2 as \( 1 \leq q \leq m \times n \). Now we have to find the \( m \times n \) most natural word pairs. The way in which these are found is described by the pseudo-code in Figure 5.3.

The algorithm starts with assigning the value 0 to a variable \( \text{sum} \). Next \( m \times n \) iterations are performed. Within each iteration we search for the word pair \((p, q)\) which has the smallest distance. This distance is added to \( \text{sum} \). The final word distance is equal to the average of the distances corresponding with the formed word pairs, which is equal to \( \text{sum}/(m \times n) \). \(^4\)

**5.2 Levenshtein distance using acoustic word samples**

Once we are able to find distances between sound samples, it is not a big step to extend the methodology so that the distances between word samples can be found as well. We do not use the acoustic representations of separate segments, but complete acoustic representations of whole words. Transcriptions are only used to find the number of segments of words. This number is used to normalize the speech rate. Kruskal and Liberman (1999), Hunt et al. (1999) and Ten Bosch

\(^4\)An alternative approach of processing multiple variants is given by Nerbonne and Kleiweg (2003).
function word_distance(W1,W2);
begin
sum:=0;
repeat
    for all possible word pairs (p,q) do
        if (Levenshtein_distance(W1(p),W2(q))<smallest) and
            p and q not used in previous formed pairs
            then smallest:=Levenshtein_distance(W1(p),W2(q))
        else {nothing};
    end;

    sum:=sum+smallest;
until m*n word pairs are found
word_distance:=sum/(m*n);
end

Figure 5.3: Algorithm in pseudo-code for finding the most natural word pairs. The algorithm assumes that the $m$ pronunciations of word $W_1$ are multiplied $n$ times and the $n$ pronunciations of word $W_2$ are multiplied $m$ times before the algorithm is called.

(2000) present methods with which pronunciations are compared on the basis of the acoustic signal.

Kruskal and Liberman (1999) describe the development of continuous time-warping and ‘formulate discrete analogues to all concepts and definitions involved’ (p. 127). A continuous function in multidimensional space is called a trajectory. For trajectories it holds that ‘variation in speed appears concretely as compression and expansion with respect to the time axis’ (p. 125). Among other things time-warping makes it possible to ‘measure how different two sequences are in a way that is not sensitive to compression-expansion but is sensitive to other differences’ (p. 125). Kruskal and Liberman ‘formalize the notion of a time-warping as a “linking” that connects the time scales of the two trajectories or sequences’ (p. 129). The distance between two trajectories is defined ‘as the minimum possible length of any linking between them’ (p. 129). The chief application of time-warping has been in speech processing which makes the methodology interesting for dialect comparison as well.

Hunt et al. (1999) present a syllable-based speech recognition system in which unknown syllables are acoustically recognized by matching them against stored
syllable templates. Syllables are represented as a sequence of acoustic-parameter vectors, each vector corresponding to one time frame. A Levenshtein algorithm finds the optimum frame-to-frame correspondence between the template syllable and the unknown syllable and calculates the distances between them over that optimum frame correspondence.

Ten Bosch (2000) describes research in which an Automatic Speech Recognition (ASR) based distance measure is used to find the acoustic distances between dialects. Words are represented as a series of frames where each frame contains acoustic features. Words are compared by aligning the frames by a Viterbi alignment procedure, a technique roughly comparable to how phonetic segments are aligned when using transcriptions. Alignment is done by matching the frames with trained ASR Hidden Markov Models (HMMs). More about the Viterbi algorithm and HMMs can be found in Manning and Schütze (1999).

The advantage of comparing words directly on the basis of acoustic samples is that no transcriptions need to be made. It is time consuming to make phonetic transcriptions and, furthermore, the quality of the transcriptions varies greatly, depending on the skills of the transcriber. In this section we present the methodology for the comparison of word samples (almost) without the use of transcriptions.

In Section 5.2.1 we describe some necessary manipulations that should be applied to the samples first. When comparing sounds, we examine several representations of the acoustic signal. The same representations are used here and discussed in Section 5.2.2. In Section 5.2.3 we explain how we normalize different speech rates. In Section 5.2.4 we describe how distances between word samples are actually found using the Levenshtein distance. These sections contain material published in Heeringa and Gooskens (2003).

5.2.1 Preprocessing

The voices of different speakers will have different pitches. Most obvious is the difference in pitch between male and female voices. Furthermore, the intonation per speaker may vary individually, in a way unindicative of variety. When two speakers read the same text aloud, the one may stress different words than the other. To make samples of different speakers as comparable as possible, all word samples are monotonized, i.e. manipulated to have the same pitch for all times in the sample. When there are male speakers and female speakers, we found the mean pitch of the men and the mean pitch of the women first. Next, all samples were monotonized on the average of the two means with the program PRAAT. Figure 5.4 shows spectrograms of non-manipulated word samples, while Figure 5.5 shows spectrograms of the corresponding monotonized word samples.

Just as for the sound samples, the volume was not normalized because volume contains a good deal of sound specific information. For example it is specific for the [v] that its volume is greater than that of the [f].
5.2.2 Representation of words

When comparing words, we do not use the type of spectrogram most commonly used which has a Hertz-scale, but the more perceptual models which we also used for the comparison of segments. The Barkfilter is described in Section 4.3.1 and the cochleagram is described in Section 4.3.2. Formant tracks represent the prominent frequency tracks in the spectrogram. In this more reduced representation speaker-specific information may be filtered away to a greater extent. The formant track representation is described in Section 4.3.3. When using these representations for the comparison of words, mostly the same parameter values are used as when comparing segments. Parameters which deserve particular attention are discussed below.

Just as in the comparison of segments forward-masking is not taken into account when using the cochleagram representation (see Section 4.3.2.1). The same word pronounced at different speech rates gives different sample sizes. When applying forward-masking (in PRAAT a default value of 0.03 seconds is given), the effect on smaller samples is relatively larger than on larger samples. This shows that forward-masking depends on speech rate. Because we suppose that speech rate is speaker-dependent, we want to reduce its influence as much as possible in the comparison of dialects. Therefore, we do not apply forward-masking.

When using the formant tracks in the comparison of segments, only two tracks are used to get results comparable to the IPA vowel quadrilateral, which also reflects F1 and F2 only. For word comparison this restriction need not to be maintained since no comparison with the IPA vowel quadrilateral will be made. The number of formants may vary over time in a word and per word. In the PRAAT program, we maintain the default value for the maximum number of formants which may be found: 5. Next, we find the minimum number of formants examining all times of all words which are taken into consideration. After that, on the basis of this minimum number of formants the word samples are compared. In the samples we use (see Section 7.2.3) for each word sample at each time sample, at least three formants could be found. Therefore, the comparison of word samples here is based on (the first) three formant tracks. Furthermore, in the PRAAT program the ceiling of the formant search should be set to 5000 Hz for males, and to 5500 Hz for females. Because the samples on the basis of which the formants are determined were monotonized to the average of the mean pitch of the males and the females (see Section 5.2.1), we set this ceiling to 5250 Hz.

To illustrate the differences between the several representations when applied to word samples, we show visualizations of three Norwegian pronunciations of the word nordavinden ‘the northwind’ using spectrograms, Barkfilters, cochleagrams and formant tracks (Figures 5.4 and 5.5). The pronunciations of the dialects of Bjugn, Halden and Larvik are given. The recordings were made by Jørn Almberg (see Section 7.2 for more details about the recordings). The pictures in Figure 5.4
5.2. LEVENSHTEIN DISTANCE USING ACOUSTIC WORD SAMPLES

Figure 5.4: Different acoustic representations of three Norwegian pronunciations of *nordavinden* ‘the northwind’. Starting from the first row we see respectively spectrograms, Barkfilters, cochleagrams and formant tracks obtained on the basis of the original samples.
Figure 5.5: Different acoustic representations of three Norwegian pronunciations of *nordavinden* ‘the northwind’. Starting from the first row we see respectively spectrograms, Barkfilters, cochleagrams and formant tracks obtained on the basis of the *monotonized* samples.
are made on the basis of the original (not manipulated) samples, and those in Figure 5.5 on the basis of the monotonized samples. The monotonized samples are used for dialect comparison.

### 5.2.3 Speech rate

When comparing word samples, we have to allow for the fact that different speech rates give different sample sizes. We perform a rough normalization by using the number of segments per word according to the phonetic transcription. Assume that the acoustic representation of a word sample consists of \( l \) spectra or formant bundles. If the number of segments of this word pronunciation according to the phonetic transcription is \( m \), and we want to represent each segment by \( n \) spectra or formant bundles, then we represent the complete word sample by \( m \times n \) elements. Changing the representation of \( l \) elements into a representation of \( m \times n \) elements is realized in two steps. First we duplicate each of the \( l \) elements \( m \times n \) times. This gives \( l \times m \times n \) elements in total. Second we regard the \( l \times m \times n \) elements as \( m \times n \) groups, each consisting of \( l \) elements, and fuse the elements in each group to one element by averaging them. The result is a representation of \( m \times n \) elements. We illustrate this by an example. Assume we have a word sample of \( l = 4 \) elements:

\[
\begin{array}{c}
1 \\
2 \\
3 \\
4
\end{array}
\]

If this word pronunciation is transcribed as a sequence of \( m = 2 \) segments, and we want to represent each segment by \( n = 3 \) elements, then we represent the complete word sample by \( 2 \times 3 = 6 \) elements. We change the representation of 4 elements into a representation of 6 elements. For this purpose first we duplicate each of the 4 elements 6 times. This gives 24 elements in total:

\[
\begin{array}{cccccc}
1 & 1 & 1 & 2 & 2 & 2 \\
2 & 2 & 2 & 3 & 3 & 3 \\
3 & 3 & 3 & 4 & 4 & 4 \\
4 & 4 & 4 & 5 & 5 & 5 \\
5 & 5 & 5 & 6 & 6 & 6
\end{array}
\]

Second we treat the 24 elements as 6 groups, each consisting of 4 elements, and fuse the elements in each group to one element by averaging them. The result is a representation of 6 elements:
In our research we chose $n = 20$, i.e. 20 spectra or formant bundles per segment. A higher value gives nearly the same results, but the computing time increases greatly. We are aware of the fact that our way of normalizing speech rate is a rough approach, but we hypothesize that it is refined enough to capture significant variation.

### 5.2.4 Comparison of words

The Levenshtein distance calculates the cost of changing one sequence or string into another. It determines how the one sequence or string can be changed into the other in the easiest way by inserting, deleting or substituting elements. A detailed description of the algorithm was given in Section 5.1. When finding the distance between different pronunciations on the basis of their transcriptions, the elements are the phonetic segments. However, when using the acoustic signal, the elements are spectra or formant bundles.

The cost of a substitution of spectra or formant bundles is the vector (Euclidean) distance between them. Assume a spectrum or formant bundle $e_1$ and $e_2$ with respectively $n$ frequencies or formants, then:

$$d(e_1, e_2) = \sqrt{\sum_{i=1}^{n} (e_{1i} - e_{2i})^2}$$

(5.1)

For the calculation of insertions and deletions we used definitions of ‘silence’. We defined a ‘silence spectrum’ as a spectrum for which the intensities of all frequencies are equal to 0. A ‘silence formant bundle’ is defined as a bundle for which all frequencies are equal to 0. This means that in absolute silence there are no vibrations.

As alternative, ‘silence’ can be defined as a spectrum or formant bundle which is sampled from background noise. At first sight this may seem to be a better approach because background noise is found in all the recordings. However in the recordings which we used (see Section 7.2.3) the background noise differs by dialect recording. Since the background noise was very low for each recording, we used no sampled ‘silence’, but ‘silence’ as defined above. Our definition of ‘silence’ approximates real ‘silence’ very closely without favoring one particular recording.

When the algorithm has calculated the sum of the operations, this sum is divided by the length of the corresponding longest alignment. The longest alignment inherently has the greatest number of matches.
When comparing two varieties on the basis of $k$ word pairs, it may appear that for one or more of the pairs for one or both varieties, no translation is given. In that case, the distance for that word pair is ignored, which is equivalent to taking the average of the distances of all word pairs for which translations in both dialects are available. Finally, in the implementation we only use one pronunciation per word.