A Lexicon Server Toolkit for Maltese

Author: Duncan Paul Attard

Supervisors: Mr. Mike Rosner and Dr. John Abela

Observer: Dr. Gordon Pace

Department of Computer Science and Artificial Intelligence

University of Malta

June 2005

Submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science I.T. (Hons.)

Abstract

This thesis implements a cooperative framework able to support the compilation and development of a computational lexicon for Maltese. As a consequence, this framework must provide all the tools needed by the user to communicate with the central server, and access the various services that are provided. Since some of the services were not expected to be developed in this project, one issue of paramount importance was the fact that the framework had to be easily extensible.

The system provides access both to the linguist and programmer in a coherent fashion. In addition, a series of tools aiding the linguist in text file processing and word list building, is also provided. Finally, a website where system development is monitored, and new user registration is permitted, was set up. In addition, this website provides the first free searchable Maltese dictionary, which apart from being built in a distributed fashion, tries to match with high quality works already existing in Maltese.

Acknowledgements

I would like to express my gratitude to Mr. Mike Rosner and Dr. John Abela, not only for their immense amount of help and insights given throughout the project, but also for the encouragement they have given me during the past months. I am aware of the difficulties I have caused, and for this I will always be grateful.

I would also like to thank my family and friends for bearing with me during this year, as well as my cousin for helping me with some issues on Maltese grammar together with the provision of books on the language.

Table of Contents

List of Figures

Chapter 1

Introduction

The task of constructing a computational lexicon, especially for a language like Maltese, is a very complex and laborious task, particularly if done manually. Expert linguists will often need to carry out the task themselves, and for a highly inflectional language like the one in question, much time is spent examining large amounts of texts to be able to categorize different words respectively. If on the other hand, this task is carried out in some automated or semi-automated fashion using well-known methods, it is very unlikely that a set of methods from one discipline will address the task in a global and appropriate manner.

The computational lexicon construction process is an extremely important task in the area of Natural Language Processing (NLP), and usually, its product is a fundamental building block of other systems, such as NL Understanding systems, parsing, spell checking, and the like. Therefore, if all other users of the lexicon services (including other systems) are to get the best of such a lexicon, its construction must not be taken lightly, as it could cripple other systems depending on it.

For a language like Maltese[1], which is a relatively 'virgin' territory, as far as NLP is concerned (Rosner et al., 1998), very few electronic resources exist. This lack of tools thus makes Maltese both hard to analyze and to use, especially in other systems, as any needed analysis of the language, must be integrated and handled in the systems in question, rather than using already existing services.

The aim of this thesis is therefore to provide this basic building block, which will hopefully not only provide a basis for other systems, but will also promote the Maltese language, in the sense that it will provide linguists with tools making the analysis of

[1]A very brief introduction of the Maltese language is provided in Appendix B on page 87.

the Maltese language possible. Of course, being a largely time consuming task, it is not expected that we develop all the necessary tools, yet, a solid framework must be constructed so that in the future, tools developed by others can be easily integrated into the system, not only to use existing services, but also provide to additional ones in a standard manner.

This project focuses mainly on two things: the building of such a framework, and also the necessary tools supporting its development, in the sense of management and user tools. The main goals of the system are itemized below:

- Provision of a solid database back-end, and a clean interface to the tools and users that will access the system services, where client programs and programmer API are to be handled seamlessly in a distributed fashion. The framework must be readily extensible.
- Provision of structure to the words in the lexicon by using knowledge-free clustering techniques which will cluster both Semitic and Romance words in a satisfactory manner.
- Creation of a basic online dictionary providing not only meaning to users but also accessible to programs wishing to retrieve information about words, such as their part of speech, origin and the like.
- The creation of a set of tools which will allow the users to use the framework in a cooperative manner, aiding in the distributed construction of a word list and dictionary for Maltese.

The system was designed in a distributed fashion, in which users are able to download the necessary tools, and then work and connect remotely to the main lexicon server. The advantages of such an approach are enormous, the most evident of which is the distribution and delegation of work not only to a 'closed' number of researchers, but to anyone that wishes to participate in the word list construction. An online accessible dictionary, aiming to reach the quality of well known works, has also been started, and can be built in such a distributed fashion.

1.1 Organization

This report has been segmented into a series of chapters aiming to provide clarity and cohesiveness; these will be outlined briefly below.

The chapter following this one presents a general discussion on the subject matter, as well as ideas that were encountered during the research period. It focuses on bioinformatics, alignment and distance finding techniques, morphology analysis together with clustering techniques. Finally, an analysis of previous work on Maltese computational linguistics is given.

Chapter three is the first in a series of two chapters discussing design, and uses the ideas mentioned in the preceding chapter to consider the problem of building the computational lexicon. Details regarding the framework, as well as the client application used to access the lexicon remotely are given, together with the advantages and disadvantages of this approach. An extensive discussion on the clustering techniques used to provide the lexicon with structure is also given.

The fourth chapter ends the discussion on design, and explains how the external interfaces (both programming and graphical user interfaces) provide access to the lexicon services. Also, the website that is used by the system to allow users to register and use these services is outlined. This chapter concludes by presenting the dictionary website which is accessible via a web browser, or programmatically by making use of the APIs provided on the site.

Chapter five discusses the results that were obtained by our clustering algorithm discussed in Chapter three, and also compares and contrasts our algorithm with others that were taken from a similar approach used in Arabic. Chapter six concludes this report by stating possible future enhancements and conclusions.

To support the material discussed in these chapters, three appendices have also been provided. The first appendix includes the test set that was used to evaluate the algorithms in Chapter five, together with a list of affixes that were used by the algorithms under evaluation. The second appendix attempts to provide an introduction to the Maltese language and grammar, highlighting the main and commonest points in the language. The third appendix provides instructions on how the system should be set up and installed on a new machine.

Chapter 2

Background

This chapter discusses the ideas which surfaced during research period, together with previous work, not only on other languages, but also on the Maltese language itself. The three main areas that influenced this thesis fall under the fields of Bioinformatics, Computational Linguistics and Clustering techniques. As the idea behind this project came from previous work[1] related to bioinformatics, it seems appropriate that these techniques would be discussed first. We then move on to the examination of previous works related to automated morphology finding, ending by a discussion of clustering techniques and a review of the work done on Maltese. The order of the subsections in this discussion is imposed by the order of relevance that these have with regards to this project.

2.1 Bioinformatics

With much research going on in the fields of molecular biology and biochemistry, the collection and sequencing of various genomes and proteins belonging to several species has been successfully completed. Yet, with the sheer amount of research that is going on world wide, tremendous data buildup is occurring, and therefore, efficient storage and retrieval of data is mandatory. Several publicly accessible databases exist[2], so

[1]This was an Assigned Practical Task last year.

[2]Examples of such databases include the DNA Bank of Japan (DDBJ), GeneBank and the Europe Molecular Biology Lab (EMBL). These three databases collaborate daily with each other, and exchange their information so that virtually, all three data banks share the same data on any given day. Some of these banks provide *submission tools* which allow a researcher to annotate sequences and submit them to these banks, where they can be later reviewed by scientists and rejected/accepted into the collection.

that gene and protein related data is well maintained, and made accessible to the public. As is apparent in subsequent chapters, some of the ideas that these databases use have been adopted in this project.

2.1.1 Sequence Alignment

Several interesting areas exist in this field but the one that we are interested in, and that is directly related to the project is that of sequence alignment. Put simply, the task of aligning a pair (or more) of sequences is to write the sequences to be aligned, one in each row, and then, by either adding gaps (denoted by '-') or allowing mismatches, try to match as many of the letters in the two sequences as possible by writing them in the same column. When sequences are found to be similar (i.e. yield an acceptable number of matches), it is more probable that these share some common characteristics, such as their structure. We now define some basic terms which will be used throughout this discussion.

Definition 2.1 (Morpheme) is the smallest meaning-bearing unit in a word.

Definition 2.2 (Affix) For our purpose, an affix is a morpheme which is glued to a stem (or word) to produce a new word with a meaning somewhat related to the original stem. We will identify two types of affixes, namely prefixes and suffixes. Prefixes are attached to the beginning of the word, while suffixes are attached to the end of the word. Infixes are also considered as affixes.

Definition 2.3 (Stem) A stem can be defined as a substring of a word that is free of its affixes. It is usually the base form for a word.

Definition 2.4 (Root) A root is a set of consonants which identify a word. In Maltese, the root of a verb is the set of consonants of the third person singular (past) which is also called the *mamma*. In the case where the mamma does not exist, the consonants of the simplest noun are taken[3]. Note that a root may not be unique, and the same root may imply a different word altogether.

The class of Semitic languages, like Arabic, Hebrew and even some parts of Maltese, relies on the concept of roots. Unlike Indo-European languages, which are stem

[3]For example, the verb *baħħar* (he sailed) does not have a mamma, but we use the noun *baħar* (sea) to extract the roots $\sqrt{\text{BĦR}}$.

based, and preserve their stem during morphology, Semitic languages only preserve the root[4] and thus, are more difficult to deal with, especially when infixes introduce new consonants even between the consonant of the root. Moreover, vowels have little meaning in Semitic languages, and these can easily change between one word form and the other. The concept of alignments can be readily applied to such words, primarily because of their fixed root patterns. The order of the root consonants is preserved during word formation, and thus, in a similar fashion to genomes, the existence of residues (which in this case are consonants) give a good indication of whether words are related or derived from the same word.

There are mainly two different types of alignments that can be performed over sequences: pair-wise and multiple sequence alignments. These will be briefly discussed below.

The Pair-wise Alignment Problem

Two ways in which a pair of sequences can be aligned exist. The first type of alignment, referred to as *global alignment*, is performed over the entire pair of sequences. The second type, which is carried out on specific regions of a given pair of sequences is known as *local alignment*. These two types of pair-wise alignment techniques are reviewed below.

We first start by discussing the global alignment problem. The aim of global alignment is to try and find out an alignment (out of the potential hundreds of others) which is optimal. Consider the two sequences `X = abcd`, and `Y = aebcde`. Two possible alignments would then be:

$$\texttt{A=}\begin{cases}\texttt{a-bcd-}\\ \texttt{aebcde}\end{cases}\qquad\qquad \texttt{B=}\begin{cases}\texttt{abc-d-}\\ \texttt{aebcde}\end{cases}$$

It is clear that alignment `A` is more optimal than alignment `B`, since there is a greater number of letter matches than in the second alignment. The job of finding out the best possible alignment is done by assigning weights to the three different situations that can occur in any given alignment. These are:

1. *Match:* a match occurs when two identical letters are aligned in the same column. Any sequence alignment algorithm should aim for the greatest number of

[4]In some Maltese words deriving from Semitic, this is not always the case (*verbi neqsin*): tar ($\sqrt{\text{TJR}}$) *flew*; tajra *kite*, where $\sqrt{}$ is used to denote the root.

matches.

2. *Mismatch:* this is when two mismatching letters are aligned in the same column.

3. *Gap:* denoted by a hyphen in the above sequences. The function of the gap is to shift the entire sequence to the left. Usually, the goodness of an alignment is determined by its high number of matches and its low number of gaps. Gaps tend to be discouraged, either by simply assigning negative weights to each gap, or else, by using different penalties for first introducing a gap, and then elongating it.

Scores to matches, mismatches and gaps can be assigned either by using a simple scoring system, or else by using a scoring matrix. This scoring matrix contains costs for the different operations related to each letter in the language being used, and therefore, on each operation taking place, this matrix is consulted, and the corresponding score is applied. Examples of such scoring matrices for protein amino acids include the Percent Accepted Mutation 250 (PAM250) by Dayhoff, or the blosum substitution matrix 62 (BLOSUM62).

The process of finding the highest alignment score can be carried out either by an exhaustive search, having an exponential complexity, or else, by using dynamic programming techniques. The intuition behind dynamic programming algorithms is that a problem can be solved by first solving sub-problems, and then combining them to yield the final solution. Usually, intermediate solutions to sub-problems are kept in a table or matrix, which the algorithm can refer to during its computation (Jurafsky and Martin, 2000, pg. 155).

The *Needleman-Wunsch* algorithm (1970) for computing global pair-wise alignments is a dynamic programming algorithm which is able to compute the optimal alignment in $O(m \times n)$, where m is the length of the first sequence, and n the length of the second sequence (Mount, 2001, pg. 72). This algorithm, which involves three steps, works as follows. It first creates and m by n matrix (this matrix/table is used for computation, and has nothing to do with the scoring matrix), and then, initializes the first row and column with zeros. Then, it progressively computes the score for each cell, starting from the upper-left corner, and gradually moving towards the bottom-right corner. Apart from computing the scores, the algorithm sets up *trace-back* pointers which will be used to trace back the computation matrix, and then, retrieve the alignment with the highest score. Note that there might be more than one alignment with the same high score. The last step would be to trace back into the computed table, and retrieve the best alignment(s).

Let us define the scoring function σ which produces the score associated with its input. Thus, $\sigma(x, y)$ returns the score associated with x and y. This score can be retrieved either by using a simple scoring system, or by searching into an appropriate scoring matrix. Also, define the score of an alignment for a pair of sequences $X = a_1, a_2, a_3, \ldots, a_m$ and $Y = b_1, b_2, b_3, \ldots, b_n$ at cell (i, j) to be $S(i, j)$. Then while the matrix is being filled, the score for each cell can be the result of either $S(i, j)$ where $a_i = b_j$ or $S(i, j)$ where $a_i \neq b_j$, in which case:

1. a_i is aligned with b_j and there is a mismatch. In this case the score at the current cell (i, j) is $S(i-1, j-1)+$ mismatch score; i.e. $S(i, j) = S(i-1, j-1) + \sigma(a_i, b_j)$.
2. a_i is aligned with a gap. In this case, the gap penalty is applied, yielding $S(i, j) = S(i-1, j)+$ gap penalty; i.e. $S(i, j) = S(i-1, j) + \sigma(a_i, -)$.
3. b_j is aligned with a gap, in which case the gap penalty is applied, giving $S(i, j) = S(i, j-1)+$ gap penalty; i.e. $S(i, j) = S(i, j-1) + \sigma(-, b_j)$.

When all three scores have been computed, the *maximum* value is retained in the current cell being processed. To summarize the above:

$$S(i, j) = max \begin{cases} S(i-1, j-1) + \sigma(a_i, b_j) \\ S(i-1, j) + \sigma(a_i, -) \\ S(i, j-1) + \sigma(-, b_j) \end{cases} \tag{2.1}$$

The trace-back pointers are set up by pointing to the (previous) cell which contributed to the highest score in the current cell in question. When the entire $m \times n$ matrix is computed, trace-back is performed starting from the lower-right corner, moving towards the upper-left corner, considering all possible alignments on the way (if the algorithm is to return more than one).

One disadvantage of the Needleman-Wunsch algorithm is that it computes the alignment of two sequences in a general, global manner, where it may be the case that there are specific regions of sequences having high similarity, suggesting that the sequences are homologous in some way or another. In global alignments, this detail is lost due to the fact that other matches or mismatches in the sequence outweigh these few regions of high similarity.

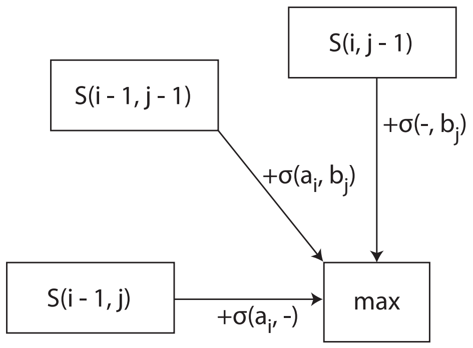


Figure 2.1: Computing the scores in the Needleman-Wunsch algorithm.

The *Smith-Waterman* algorithm (1981) addresses this problem by aligning a region from one sequence with another region in the other sequence (Mount, 2001, pg. 73). Like the Needleman-Wunsch, it is based on dynamic programming, and has the same space and algorithmic complexity. As its name implies, the output of the algorithm is not an alignment involving all the two sequences being analyzed, but rather an alignment between certain regions of the sequences.

The Multiple Alignment Problem

The pair-wise alignment problem can also be extended to cover more than two sequences. The multiple alignment problem is very similar to the pair-wise alignment problem, since we can also calculate the score of a sequence using the σ function which takes a variable (more than two) number of parameters. Thus, if for example, we align four sequences A, B, C, D, the σ function will be $\sigma(a_i, b_j, c_k, d_l)$.

When calculating the scores of a pair-wise alignment, there are four possible ways of obtaining the score of a cell, namely, $\sigma(a_i, b_j)$, when a_i matches or mismatches b_j, $\sigma(a_i, -)$ when a_i matches a gap, and $\sigma(-, b_j)$ when b_j matches a gap. In a similar fashion, this process has to be extended to multiple alignments, and clearly, with four sequences there are 16 (2^4) comparisons to make. In essence, the algorithm displays an algorithmic complexity, and therefore, as the number of sequences and their length increases, the problem becomes unfeasible to solve. Usually, heuristics are employed to approximate to a satisfactory solution.

2.1.2 Distance Measurement

Sequence alignment and distance measurement are two closely intertwined concepts, and are often used interchangeably. In this report, we will refer to them as *sequence comparison*. The result of sequence comparison is either optimal distances or optimal comparisons, like alignments for instance (Sankoff and Kruskal, 1999, pg. 31). A relatively simple application of distance would be for example to judge whether two protein sequences or genomes share the same common ancestor. If say, the distance between them is small enough, where the meaning of ‘small’ can be bound by some constant, then one might say that they are homologous. Of course, large distances between sequences do not imply that these are non-homologous, but that maybe their relation is obscured by the fact that they may have had common ancestry in the distant past. In a similar fashion, a small distance between two given words would imply that they either share parts of the stem or of the same root; large distance would either mean that the words are orthographically unrelated, or else the morphology (analogous to mutation) is so complex that any evidence of similarity is almost wiped out.

To aid our discussion in relation to distance measurement, we will define the following:

Definition 2.5 (Metric) Usually in mathematics, distance, denoted by the function d satisfies the *metric axioms*

1. Non-negative property: $d(x, y) \geq 0$ for all x and y.

2. Zero property: $d(x, y) = 0$ if and only if $x = y$.

3. Symmetry property: $d(x, y) = d(y, x)$ for all x and y.

4. Triangle inequality: $d(x, y) \leq d(x, z) + d(z, y)$ for all x and y.

Although we are interested in the above properties, some distance functions do not necessarily obey all of them.

There are several types of distance functions one might use, some of which are more commonly known:

1. Euclidean distance in 2-D: $d_e = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$

2. City block distance in 2-D: $d_{cb} = |x_1 - y_1| + |x_2 - y_2|$

3. Hamming distance, defined for strings of the same length. For two strings X and Y, the hamming distance is the number of places in which those strings differ.

4. Levenshtein distance, defined for strings of arbitrary length. It counts the number of insertions, deletions and substitutions that are needed to change one string into another.

Ordinary variables which characterize a single unit of study, such as for example, the weight of a person, or the length of a string, are called *monadic* variables, whilst variables characterizing pairs of units of study, such as the difference in weight between two persons, or the difference in length between two strings, are called *dyadic* variables (Sankoff and Kruskal, 1999). Some methods of analysis, such as clustering algorithms, require dyadic input, and therefore, distances measures like Levenshtein distance are useful in evaluating the distance between two given strings. In this section we concentrate on this type of distance primarily for the fact that we are interested in the distance between two strings. However, as we shall see, other measures of distance are employed in subsequent chapters.

Levenshtein distance is an instance of a *minimum edit distance* algorithm, and in its simplest form uses the weights: 1 for insertions and deletions, and 2 for substitutions. In this version, substitutions can also be disallowed since a substitution can be achieved by a deletion followed by an insertion. Alternatively, instead of using fixed weights, one can use a weight matrix similar to the one discussed in section 2.1.1. The advantage of using a weight matrix is that we are allowed to specify different weights for different operations. For example, in Semitic languages, consonants play a much important role than vowels or weak consonants. Therefore, we might want to give the former larger weight than vowels or weak consonants. This type of Levenshtein distance, where different costs are associated to the several editing operations is known as *weighted Levenshtein* distance (Fred and Leitão, 1998).

The basic Levenshtein distance algorithm, which is based on dynamic programming, makes use of a table or matrix to keep its intermediate calculations (once again, this is independent of the weight matrix). Similar to the sequence alignment algorithm, it is able to compute the minimum edit distance in $O(m \times n)$; m being the length of the first sequence, and n, the length of the second sequence.

Let X and Y be two sequences where $X = a_1, a_2, a_3, \dots, a_m$ and $Y = b_1, b_2, b_3, \dots,$

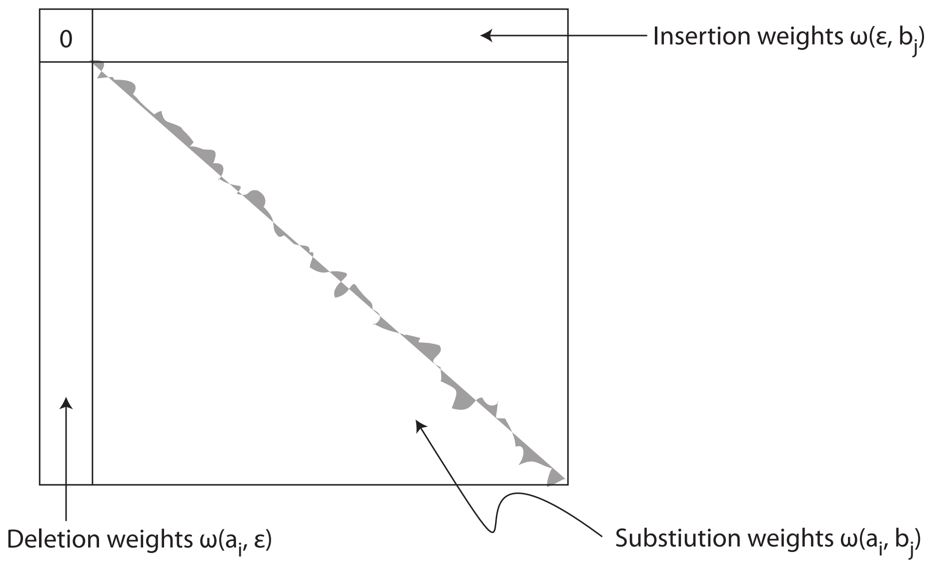

0
Insertion weights ω(ε, b_j)
Deletion weights ω(a_i, ε)
Substiution weights ω(a_i, b_j)

Figure 2.2: The weight matrix that can be used to weight the various operations used in calculating the distance. A similar matrix can be used in the pair-wise alignment algorithm. If the distance function is to follow the *zero property* of the metric axioms, the values on the scribbled diagonal should be set to zero.

b_n. We will call X the target string and Y the source string[5]. Also, let us define the distance at a given cell (i, j) to be $D(i, j)$. In a similar vein to the σ function, the ω function for x and y can be used as follows:

1. a_i is substituted with b_j. In this case, the score at the current cell (i, j) is $D(i-1, j-1)+$ substitution score; i.e. $D(i, j) = D(i-1, j-1) + \omega(a_i, b_j)$.

2. a_i is deleted from target. In this case, the deletion score is applied, yielding $D(i, j) = D(i-1, j)+$ deletion score; i.e. $D(i, j) = D(i-1, j) + \omega(a_i, \epsilon)$.

3. b_j is inserted into source, in which case the insertion score is applied, giving $D(i, j) = D(i, j-1)+$ insertion score; i.e. $D(i, j) = D(i, j-1) + \omega(\epsilon, b_j)$.

Note that the empty string is denoted by ϵ in the above. When all three have been computed, the *minimum* is taken and assigned to the cell being processed. To summarize:

[5]The source sequence is the string we are changing (by using the insertion, deletion and substitution operations) into the target string. In our explanation, the source string is the placed along the side of the distance table, and the target is placed along the top.

$$D(i,j) = min \begin{cases} D(i-1,j-1) + \omega(a_i, b_j) \\ D(i-1,j) + \omega(a_i, -) \\ D(i,j-1) + \omega(-, b_j) \end{cases} \tag{2.2}$$

Similar to the sequence alignment algorithm, it involves three steps (the third step is optional), the first one of which involves creating the m by n matrix used for computation, and initializing its first row and column by using the weight scores respective to the source and target strings. Initialization for the source is done by computing $D(0,j) + \omega(\epsilon, b_j)$ for $j = 0$ to n. Similarly, for the target sequence, initialization is done by computing $D(i,0) + \omega(a_i, \epsilon)$ for $i = 0$ to m. Note that for this purpose, an extra row and an extra column are added so that the empty string ϵ is included. Thus, the distance matrix is actually $[m+1, n+1]$.

The computation step is done by starting from the top-left corner $(0,0)$, and gradually moving towards the bottom left corner (m,n). If the third step is to be implemented, then trace-back pointers must be set up so that when computation is over, the list of operations (insertions, deletions and substitutions) can be retrieved. Pointers are set up by pointing to the (previous) cell which contributed to the lowest score in the current cell in question. Trace-back is then performed in a similar way to that of pair-wise sequence alignment. Note that the distance between the source and target sequences is contained in cell (m,n).

2.2 Morphology

Although in this project, we do not specifically aim to examine the morphology of individual words of a language as whole, there are some techniques about the subject that have been employed in the project, and therefore, it is appropriate to give a general overview of the work that has been done in this field. The techniques that are of interest to this project involve automated methods of morphology analysis, and therefore, literature which did not have this objective was not taken into consideration, and as a consequence, will not be mentioned here. This was done because firstly, one of the project aims was to provide an automated means of extracting affixes from a piece of text, and secondly, there are presently no electronic dictionaries or thesauri which could assist us in our task, thus this was our only feasible option.

The construction of a morphological analyzer for the language in consideration

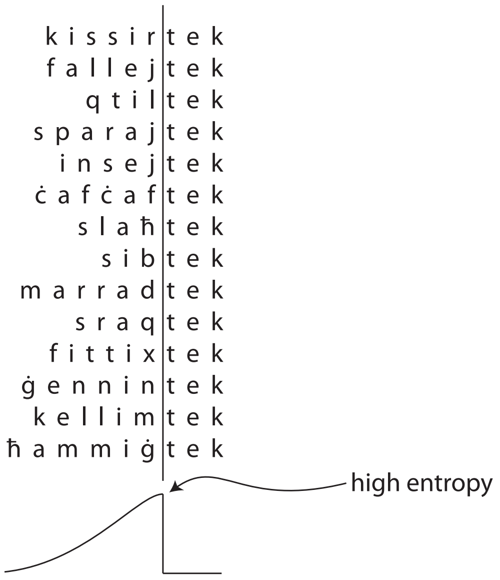


Figure 2.3: The entropy measure allows the Harris' algorithm to detect the presence of morpheme boundaries.

normally involves considerable amount of work by expert linguists. Most often than not, this process is expensive, time-consuming, and even not applicable to all languages; this normally requires us to make certain assumptions about the language we are dealing with. For example, if we are dealing with English, our assumption would be that a word consists namely of a prefix, a stem and a suffix. If on the other hand, we are dealing with agglutinative languages such as Turkish, then a stem may be followed by more than one suffix. Mixed languages like Maltese may either involve the use of a stem (if the word is of Romance origin) or a root (if the word is of Semitic origin). Luckily for Maltese, both semitic and romance words usually use the same set of affixes when verbs are being conjugated (Aquilina, 1987-1990). The two types of morphology analysis that we came across are presented concisely below.

2.2.1 Morphemes over a Language

A first approach to analyzing morphology automatically was proposed by Zellig Harris in 1955. The idea behind his approach was to identify the morpheme boundaries leading indirectly to the discovery of stems. Distinguishing this method from the rest was the fact that it made use of corpora, and also employed the idea of entropy,

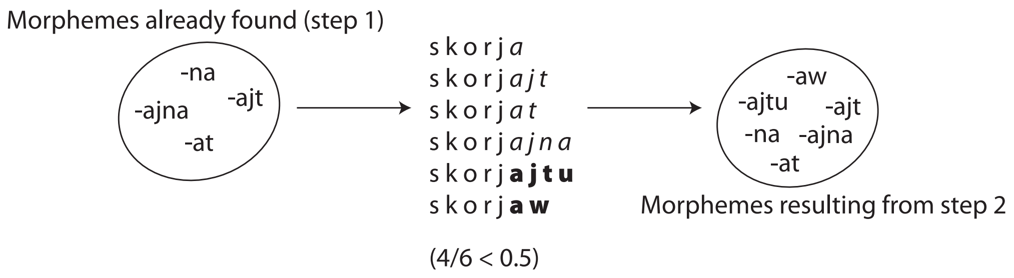


Figure 2.4: The algorithm proposed by Déjean. The suffixes in italics denote those found in step 1. The suffixes in bold are those that will be accepted, since the words containing suffixes from step 1 amount to more than one half (4/6) the total words presented.

that is, the number of different letters that may appear after a given word segment (Fig. 2.3 on the previous page). The increase in entropy for a given letter indicates a probable morpheme boundary. This algorithm works reasonably well providing the corpus being used contains enough variants of the same stem. Indeed, wrong segmentations can be generated.

Another approach taken by Déjean, which was inspired by Harris' algorithm aims to find the morphemes of a language (like Harris) rather than discover word-related morphemes. The idea is to improve on Harris' algorithm by dividing the word segmentation process into three separate phases. The first phase involves applying Harris' algorithm, retrieving a list of the most frequent suffixes[6], and then filtering this list by using a threshold of 100 (Déjean, 1998). The second step further augments the first list of suffixes by segmenting the words with the help of the suffixes already generated in the first step. These are obtained by examining a given sequence of letters, and if the remaining sequences of letters (from this point till the end of the word) correspond to the morphemes already found, and the number of matches exceeds half the morphemes found in step 1 (suffixes in this case), then the others are also accepted as morphemes (Fig. 2.4). The last step involves segmenting the words by matching the longest possible suffix under the assumption that the longest suffix contains one or more morphemes[7] and therefore, is the most appropriate.

[6] Although suffixes are being mentioned here, the algorithm can be easily adapted to prefixes by reversing the letters of a word.

[7] In fact, this algorithm does not generate 'split' morphemes, and therefore we would expect to find such suffixes as '-ment' and '-ments', instead of 'ment.s' where the period signifies a further morpheme boundary.

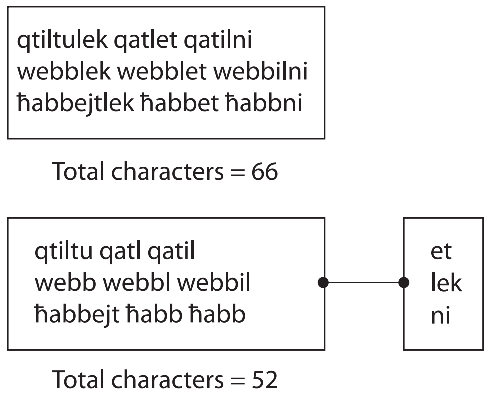


Figure 2.5: An example of MDL, where by finding an appropriate set of suffixes, the description length was reduced by 14 characters.

2.2.2 Morphemes over words

Whereas in the previous subsection we have discussed algorithms that enable us to find suffixes (or prefixes) over the language as a whole, in this section we consider more advanced algorithms able to associate sets of affixes with words. The first two methods that are mentioned employ the notion of *Minimum Description Length* (MDL). The idea behind this approach is to try and compress as much information into the smallest possible description; similar to the way in which learning takes place in humans, where out of a potentially infinite amount of information, the brain is able to encode in some compact way this information, and usually the more the number of generalizations, the better the learning. Fig. 2.5 illustrates the idea behind this concept.

One such method making use of the MDL concept is discussed in Goldsmith (2001), in which he proposes a statistically, MDL-based, knowledge-free method able to learn the morphology of words in a given corpus of text, even as little as 5000 words. This method is aimed at Indo-European, particularly those languages which are 'stem + suffix'-based. The main idea behind this model is to search iteratively for the morphology with the smallest description length.

Bootstrap heuristics are used to give the algorithm an initial hypothesis of what stems and suffixes are, and then by using these, start an iterative search, looking for the morphology which yields the smallest description. Two bootstrap heuristics are discussed, the first one of which fits a Boltzmann distribution to the probabilities of each of the n cuts that can occur in a word having n characters. This augmentation of probabilities was done so that long suffixes and stems are favored over the shorter ones.

The second heuristic originates from the fact that the algorithm is targeted towards the identification of suffixes. Starting from the back of a word, and under the assumption that no suffix contains more than five characters, n-grams for $2 \leq n \leq 6$ are extracted[8]. Then, by rating these appropriately, the top 100 are chosen as the set of candidate suffixes. After bootstrapping, the algorithm computes the cost of the morphology in question, and if it is found out that the latter is more compact than the previous morphology, the most compact one is retained and the other discarded; in this way the model is optimized gradually.

Goldsmith makes use of the notion of *signatures*, which are structures composed from a stem followed by an alphabetized list of suffixes that the stem can take. Two lists, one containing stems and the other containing suffixes, are maintained, and using pointers, signatures can be appropriately composed by pointing to stems and suffixes from the lists mentioned. This structuring of signatures is recursively defined, so that suffixes containing other suffixes are properly stored[9]. Incremental heuristics are used during the morphology search, and these include discarding signatures appearing with only one stem, or shifting the stem-suffix boundary to the left.

The work due to Kazakov (1997) is another technique based on MDL, also falling under the class of unsupervised learning. However, this method relies on *Genetic Algorithms* (GAs) to produce an appropriately evolved morphology. The motivation behind this is that

> "A large search space and the lack of applied algorithms are good premises for the use of GAs." (Kazakov, 1997)

For the algorithm to be able to process words, morpheme boundaries are encoded into integer vectors, where each integer in the vector corresponds to an index where the morpheme boundary is located in a word. These vectors are directly used as chromosomes and fed into the GA. The mutation operator for this purpose was defined as either a shift of the morpheme boundary to the left or right, or as a random choice of

[8]In this heuristic, a word is assumed to end with the stop symbol '#'. Thus 'hello' would be written as 'hello#'.

[9]This phenomenon is extremely common in Maltese, like for example: ksirthielux = ksirt (broke, past.) + hie (her, 3rd. pers. sing. fem. obj.) + lu (his, 3rd. pers. sing. masc. subj.) + x (negation), meaning "I did not break his (fem.) thing".

the boundary in an allowed interval. In a similar manner to Goldsmith, the goodness of the morphology is estimated by the number of characters that are involved — the lesser this number, the more compact the morphology is.

A last technique that shall be discussed attempts to analyze the morphology of Assamese — a major Indian language of the Indic branch of the Indo-European family (Sharma et al., 2002). The technique used here, although different from that used so far, employs the idea of *characteristics*, which is somewhat similar to the concept of signatures used by Goldsmith. In any set of words, there are some cases in which one word can be derived from some other word in that set. This is defined by a *decomposition*, where a word derivative is made up of the base (analogous to stem) and the rule (suffix). Clearly, different word derivatives can be created by changing the rule for a given base. A base may also be recursively defined in the sense that it may well be a derivative with respect to another base[10].

The method for obtaining the characteristics of a word is split into one preprocessing phase and three steps. The preprocessing involves reading the input text and sorting it into a list. Then, for each word w in the list, the algorithm tries to identify a base b and a suffix s so that $w = b + s$. If such a word is found, then the word is decomposable, else it is marked as not decomposable. For each word obtained, all those decompositions containing b as a base are counted. Likewise, all those decomposition containing s as a suffix are also counted. Then, the following filtering rule is applied: decompositions having a rule with a very low value (typically 1), and bases with a very high value (these usually signify short bases which are often part of a suffix) are discarded.

The second phase considers those words which were not decomposable in the first phase, and tries to use the rules identified in step one so that potential derivatives of derivatives are identified. In the last phase, the words in the list and their corresponding suffixes are combined to form the word characteristics. With more bases obtained from the second phase, the first and second phases can be repeated in order to decompose the words further, until their base form is reached. Since in this language, and in many others, suffixes can contain other suffixes within, preference is given to the longer suffixes when a decomposition takes place. This way, the longer suffixes can gradually be reduced to shorter ones, where the longer suffixes are composed of a series of 'sub-suffixes'. Even though during word decomposition, the aim is to reach

[10]Goldsmith also employs a similar recursive definition for signatures.

the base forms, it is apparent that

> "an analysis based simply on the detection of presence of common substrings may fail to detect decompositions where the spelling of the derived words is not simply a concatenation of the base word but a modification of that." (Sharma et al., 2002)

This is very true not only for Assamese, but especially for Maltese where all Semitic-based words exhibit a high amount of inflexions, primarily because of *verb forms* as well as *weak verbs* and other additional infixes. Unfortunately, this method does not specify how to deal with such cases.

2.3 Clustering

In this project, clustering is of significant importance and interest since one of the tools that must be provided to the linguist is that of clustering a set of words[11]. Indeed, clustering is a powerful method, namely because it is fully (or almost) unsupervised, and usually partitions data into similar groups without any preconceptions on the structure of the data. Two major flavors of clustering techniques exist, namely *Hierarchical* and *Partitional* clustering; these are in turn discussed below.

2.3.1 Hierarchical Clustering

Hierarchical clustering refers to the process of creating a hierarchy, that is, organizing data into large groups which in turn, contain smaller groups, and so on. The output generated by such an algorithm can be easily drawn as a dendrogram or a tree, where the most general grouping is situated at the top of the dendrogram, and the specific grouping at the bottom. In the most specific grouping, each cluster contains only one element. Between these two extremes, one would find different groups, ranging from the most specific to the more general. One feature of this clustering method is that an element can indirectly be a member of another cluster, whereas in partitional clustering, an element can only be assigned to one group (Fasulo, 1999).

The dendrogram can either be formed by constructing it bottom-up or by starting from the top, and proceeding in a top-down fashion. The latter is often called

[11]One of the aims of the clustering facility was to allow the the creation of clusters related to word stems or roots, which can later be incorporated into the dictionary so that it would be structured in a similar way to Aquilina's dictionary.

divisive while the former is known as *agglomerative*. The algorithm for agglomerative clustering is given below:

The Agglomerative Clustering Algorithm

1. Start with n clusters, each containing one element.
2. Find the most similar clusters C_i and C_j (see below) and merge them together. If there is a tie, merge the first pair found. This step should be repeated $n-1$ times.

There exist different measures that can be used to calculate the similarity between a pair of clusters, but the most common types include the *Single-Linkage*, *Complete-Linkage*, *Average-Linkage* and *Ward's* algorithms. In our work, we also used a simple algorithm measuring the distance between the cluster centroids. The single-linkage measure is defined to be the smallest distance between two elements such that one element is in cluster C_i and the other in C_j. Formally,

$$D_{sl}(C_i, C_j) = \min_{a \in C_i, b \in C_j} d(a, b), \tag{2.3}$$

where $d(a, b)$ denotes the distance between the elements a and b (see Subsection. 2.1.2 on page 10). The complete-linkage algorithm is defined as the largest distance between an element in C_i and an element in C_j:

$$D_{cl}(C_i, C_j) = \max_{a \in C_i, b \in C_j} d(a, b). \tag{2.4}$$

While the single-linkage method states that two clusters C_i and C_j are similar if the distance between $a \in C_i$ and $b \in C_j$ is small, thus taking only one similar pair (a, b), the complete-linkage requires that all pairs in C_i and C_j are similar. The average-linkage[12] algorithm tries to compromise between these two extremes by taking the average distance between a point in C_i and a point in C_j:

$$D_{al}(C_i, C_j) = \frac{1}{|C_i|\,|C_j|} \sum_{a \in C_i, b \in C_j} d(a, b), \tag{2.5}$$

[12]This is also known as the Unweighted Pair-Group Method using arithmetic Averages (UPGMA).

2.3.2 Partitional Clustering

The goal of partitional clustering (known also as k-clustering) is to induce partitions in a set of data, where each partition contains data that is closely related. This process takes the form of a divisive strategy, where a chunk of data is gradually split to form a number of clusters. In these algorithms that are discussed below, the number of clusters must be specified in advance.

Forgy's Algorithm

One simple iterative algorithm that can be used for such a task is *Forgy's* algorithm. This takes k points as its input, thus, yielding k clusters after the algorithm terminates. Seed points can either be chosen arbitrarily, or with some knowledge of the cluster elements (if possible). Alternatively, one can use an agglomerative clustering algorithm to generate the initial seed points. Since the algorithm may take a long time to converge and produce stable clusters, some versions of the algorithm given below allow the user to restrict the number of iterations.

Forgy's Algorithm

1. Initialize by making the seed points the cluster centroids.
2. Find the nearest cluster centroid for each object, and assign it to the cluster in question.
3. If no object changed its cluster, stop, otherwise compute the centroids of each resultant cluster and go to step 2.

The k-means Algorithm

As its name implies, the k-means algorithm takes as its input k seed points together with the data to be clustered. This algorithm differs from Forgy's in that it clusters the data in only two passes, without further iterations. However, the cluster centroids are computed as soon as an object joins a cluster. Its three steps involve:

k-means Algorithm

1. Initialize the clusters with a seed point each. These seed points are the first k samples of the data. (Alternatively, they can also be specified separately).
2. For *each* remaining object $(n - k)$, find a cluster centroid nearest to it, and assign it to that cluster. After each object is assigned, re-compute the cluster centroid.
3. Go through the data a second time, and for each object, find the nearest cluster centroid to it. Assign this object to the cluster *without* computing the cluster centroid this time.

Both the k-means and the Forgy's algorithm have the same goal — that of reducing the square error for a given number of clusters. In order to achieve reasonable computation time, not all possible cluster combinations are considered, and for this reason, they may end up at local minima. Moreover, the choice of seed points, as well as the order in which they are read dictates the outcome of the resulting clusters.

One of the critical points on clustering is its way in which distance between elements is evaluated. For our application, finding the distance between strings using the Levenshtein distance measure is good but not enough, primarily because related stems having different prefixes and suffixes will most likely not be classified as related. A simple improvement would be to first perform some sort of stemming, and then try to re-cluster. Clearly, this is more likely to work, especially for stem based languages such as English, Latin and so on. However, for Semitic-based languages such as Arabic and Maltese, this refinement is not enough for the fact that highly inflectional morphology is involved in word and verb construction.

Clustering of Arabic Text

A clustering algorithm which is presented in de Roeck and Al-Fares (2000) deals with the problem of clustering Arabic words. The idea behind this algorithm is based on an algorithm presented by Adamson and Boreham (1974) which calculates the *similarity*

String	**Bi-grams**	**Unique Bi-grams**
fexfex	fe ex xf fe ex	fe ex xf = 3
tfexfex	tf fe ex xf ef ex	tf fe ex xf = 4
Shared unique bi-grams		fe ex fx = 3

Table 2.1: The bi-gram similarity algorithm. The similarity coefficient is calculated by $SC = (2 \times \text{number of unique bi-grams})/(\text{sum of unique bi-grams in each string})$, which in this case is $(2 \times 3)/(3 + 4) = 0.86$. (Adapted from de Roeck and Al-Fares (2000)).

between two strings by considering the number of shared substrings[13]. Note that now care must be taken to delineate similarity from distance. As opposed to distance, the greater the number returned by the similarity function, the more those strings are closely related. Table 2.1 outlines how the similarity function works.

Although the algorithm presented by Adamson and Boreham outperforms the conventional distance measures, since it takes also the roots and infixes into account, it is still misled by a number of factors which are listed below. The algorithm presented by de Roeck and Al-Fares (2000) tries to deal with these issues:

1. **Affixes and light stemming**. The SC is kept low due to the fact the prefixes and suffixes outweigh the root information, and therefore, words deriving from the same root but having totally different suffixes will not be clustered together. This is especially true for Arabic words that tend to be very short, where most probably, words with a different root but same affixes will be clustered together. To get around this problem, stemming is done, but it is lightly executed since care must be taken so that no root consonants are removed in the process.

2. **Weak letters, infixes and 'cross'**. The technique referred to as 'cross' is used to deal with the interference that infixes and weak letters cause. For each weak letter, this technique adds a new bi-gram containing the letter occurring before and after the weak letter. This allows the roots to contribute fully towards the SC.

3. **Suspected affixes and differential weighting** is used to give low weight (0.25) to those bi-grams containing weak letters, and to those bi-grams (0.5) containing suspected non-weak letter affixes. All the other bi-grams are given a weight of 1.

[13]More specifically, these will be bi-grams in this algorithm.

4. **Substring boundaries** is a common technique used when dealing with string comparison. Here it attaches a '*' at the beginning and end of a given word in order to allow the boundary letters contribute fully to the SC.

5. **SC formula**. The formula used in Table 2.1 gives the importance to the shared unique substrings between words by doubling their evidence. However, since in Arabic the words stemmed in point 1 will usually amount to three or four characters, the impact of shared substrings will be high. To reduce this effect, the SC formula is changed to $SC =$ (shared unique bi-grams)/(sum of unique bi-grams in each string - shared unique bi-grams).

The single-linkage clustering algorithm was used to cluster words, and threshold of α was applied so that formed clusters only contain those elements having a similarity score grater than α. Clearly the lesser the value of α the lesser clusters will form — however these will contain a large number of elements, and possibly a large number of words which are incorrectly clustered. On the other hand, a high value of α yields a greater number of clusters which will contain fewer and more related elements. Moreover, when α is high, there is the possibility that a large number of single word clusters is created.

2.4 Previous Work on Maltese

The aim of this project is to come up with a framework similar to the one discussed by Dalli (2002), that is, the creation of a framework which is both extensible and at the same time usable by linguists. A second objective is that of creating a first online Maltese dictionary, in a similar style to that of Aquilina (1987-1990). Although the latter dictionary is extremely detailed, and sought by many academics, including law students, its main problem is that it is rather a 'screen shot'[14] of the language in those days. Clearly, with the sudden growth of the media, Internet, wireless systems, new styles of living, and lately, the introduction of Malta in the E.U., the language that once was, has changed, and is constantly changing. New loan words are constantly being introduced into the Maltese language, and therefore, a dictionary on paper, although useful, is simply not enough. The idea of an evolving lexicon and dictionary is more appealing, namely because it is constantly monitoring the changes in the language.

[14]The last known edition of this dictionary was published around 1990.

The *Maltilex* project aims to provide an implementation of a computational lexicon for Maltese, where information about words and their different word forms can be readily stored, updated, and used by linguists. As is clearly stated in Rosner et al. (1998), constructing the lexicon from a dictionary is quite impossible for Maltese, because first of all there exist no such electronic artifacts, and also because different data descriptions are adopted by different linguists and dictionary writers; this requires an extra effort to integrate such disparate descriptions into a single common one.

One solution to these problems would be to first define a common framework, and then, starting from scratch, collecting as much text as possible to create an initial word list, which can later be structured in clusters using *headwords* with associated lexical items. A headword can be either seen as the representative element of a cluster or as a sequence of characters which identifies a set of related lexemes. It need not be a proper word however. For example, in Aquilina, the headword is either the *mamma* (3rd. pers. sing. past. masc.) for Semitic verbs or the stem for Romance word. Where a mamma does not exist[15], some other form of noun (or verbal noun) or variants of the first verbal form are used. For an introduction to the Maltese grammar refer to Appendix B on page 87.

Clearly, the best way to proceed would be to resort to automated methods of data processing, since the examination of a large proportion of words by hand is infeasible. However, these automated tasks can be coupled with manual work to further refine the output of such algorithms. One of the aims of this project is that of providing the tools needed to facilitate the manual work needed to be carried out by expert linguists.

Both Rosner et al. (1998) and Micallef and Rosner (2000) discuss the data representation used by the Maltilex project. In addition, Rosner et al. (1998) also describes a template used to annotate words with information about their category, part of speech, origin, etc. Dalli (2001) discusses the advancements made in the field of data representation standards such as Unicode, and also presents the modified Latin letters used by the Maltese language[16]. In our work, we will not use the Standard Maltese Text Representation (SMTR) since most modern programming languages and databases already come equipped with Unicode support, and therefore, being

[15]This is very common where the 3rd. pers. sing. past. masc. does not exist, like for example, bakar ($\sqrt{\text{BKR}}$, to wake up early). In this case, a most probable headword would be *bakkar* (he woke up early), which is the second form of the word *bakar*. Note that still, the root remains the same, i.e. BKR.

[16]Maltese is the only Semitic language using Latin characters.

easier to handle and to work with, the UTF-8 encoding will be used. Unicode is also supported by major software vendors [17].

The first known approach taken to create a computational lexicon for Maltese was taken by Dalli (2002). This work, which introduces a technique called Lexicon Structuring Technique (LST) uses automated means of clustering without knowledge of the language whatsoever. LST relies on the phonetic transcription of words as its main medium of processing. This it does, because first of all there are fewer inconsistencies between related word forms than their orthographic counterparts. Secondly, if an international phonetic language (such as IPA) is used, the application is made as general as possible.

The algorithm uses a set of weights to calculate the Euclidean distance between a given phoneme against any other. An incremental clustering algorithm is used, and clustering is performed directly on the database, in the sense that a word is dispatched for clustering as soon as it enters the lexicon. Pre-, post- and adaptive optimizations are used to gradually enhance the results produced by the clustering algorithm. Pre-optimization involves ordering the list of words being fed to the algorithm by their similarity, where least similar items are presented first. Post-optimization allows the user to edit the clusters created, and at each edit, rules related to those clusters being merged or split are generated. These in turn allow the clustering phase to refer to them to fine tune the clustering process; this is the adaptive optimization phase. A special buffer is maintained by the system so that words that are dubious are not clustered — instead they are placed in the buffer until there is enough evidence that allows them to be placed in a cluster.

The technique used to elect headwords (or representative elements) from a cluster is based on the idea of multiple alignments (see subsection 2.1.1 on page 9). The idea is to first align all the elements in a cluster, and then elect a headword containing the aligned letters of that cluster which are above a certain threshold τ. When either a new word joins the cluster, or modification due to the linguist intervention occurs, the elements in the cluster in question are re-aligned to re-elect a possible new headword.

Similarly to this project, the system by Dalli provides a set of services to linguists, and provides for the addition and deletion of words from the lexicon through an appropriate web interface, providing linguists are subscribed with the system. In addition, core and extended APIs are provided to the programmer who wants to

[17]It must be duly noted that the UTF-8 encoding did not exist in 1998, thus the SMTR was used to deal with the problem of encoding the special Maltese characters.

interface with the system and make use of its services. Other services, such as an on line tokenizer and Maltese phonetic IPA transcription are also provided[18].

[18]These can be found at `http://mlex.cs.um.edu.mt/lexicon`.

Chapter 3

Core Architecture

This project aims to provide a generic and sound framework which is practical and can be used to create a computational lexicon for the Maltese language. The functionality and design decisions that had to be taken are discussed in fair depth, and for this reason, two chapters are dedicated to the design of the system. Since the actual step-by-step implementation of the system is superfluous, we will only highlight those implementation details which are of considerable importance, otherwise, no reference to specific details is made. This first chapter on design deals with the core functionality of the system, as well as the ideas and motivations behind certain decisions. At the end, the clustering function that is provided by the system is also discussed.

3.1 Framework Design

Taking a rather practical approach, the first major design issue that was faced in the early days of the project was the programming language to be used, as well as what additional tools would be needed so that the project would be developed in a timely fashion. Taking into consideration the fact that a lot of string (mainly Unicode) processing would be done, a language which handled Unicode efficiently was in need. C# was the first language of choice, namely because of its Unicode support, as well as its ability to scale well with the size of the application, in the sense that modules and dynamic link libraries (DLLs) can be easily developed. Moreover, the use of an appropriate IDE like Visual Studio helped a lot during the framework implementation. Since the resultant artifact had to be a desktop application, languages like Java or C which offer relatively lesser help when designing Windows Forms applications (as opposed to Visual Studio) were not considered. Also, thanks to the .NET Framework which wraps around COM components, using such components is more straight forward than in Java. Moreover, .NET assemblies can also be used by Win32 applications easily.

The use of text files for storing data was out of question, and therefore, an appropriate database server was needed. Again, the selection of such a database was largely affected by the fact that the system will mainly deal with Unicode strings. Being able to meet our requirements, the popular open source MySql database was used. This database also comes equipped with an efficient data connector written specifically for C#.

3.1.1 Client Application

The idea of having a stand-alone application, rather than a public web interface was partially taken from the sequence submission model used by some of the major sequence databases[1]. The aim is to provide a relatively simple application which is able to function in disconnected mode, away from the central database. However, if sequences need to be annotated or submitted to some sequence database, then this must be carried out by e-mail. Thus, the job of the submission tool is that of transforming raw sequence data into appropriate database records, ready for submission. Staff on the other side would then retrieve such record from their mail, and verify whether a given record is fit to be added into the database.

Being able to support such a large network of researchers worldwide, the idea of this model was incorporated into our framework with some modifications. The first headache would be to use e-mail as a submission medium, when more innovative approaches can be used. E-mail is cumbersome, and apart from being inefficient and unreliable, extra burden would have to be put on the person(s) verifying submissions made to the central lexicon database. Therefore, the database is also used as an intermediary, storing submissions and amendments from clients, which in turn could be reviewed by a qualified person before being admitted into the database (Fig 3.1).

Several advantages are gained when using this framework, as opposed to web interfaces[2]. These are summarized as follows:

[1]One such tool is called *Sequin*, provided by the National Center for Biotechnology Information (NCBI), and is compatible with GeneBank, EMBL as well as DDBJ.

[2]Web interface is used to mean website. These two terms will be used interchangeably.

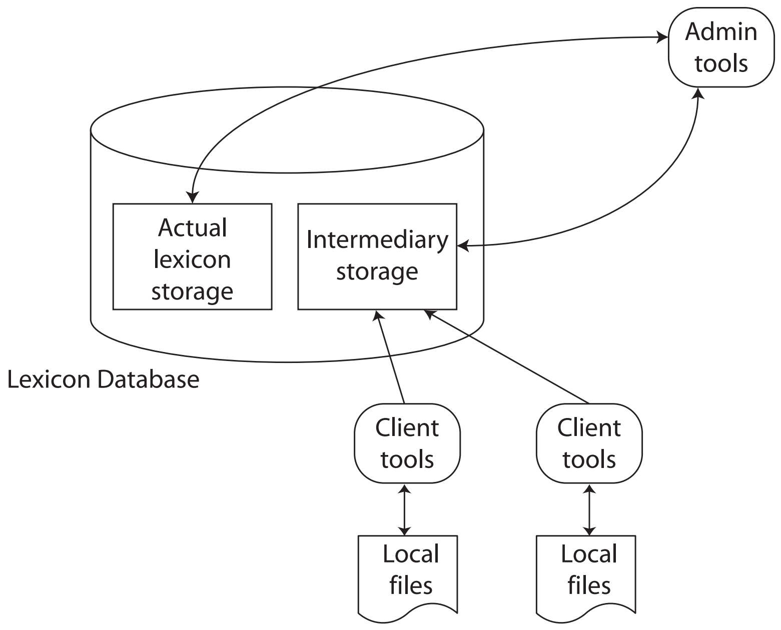


Figure 3.1: How the system uses the intermediary storage to disallow direct modification of the database by clients. Note that the client software does not include the code for the administrator functions, and vice versa. This is for added security.

1. Operating in a disconnected and standalone manner allows the user to make updates and annotations locally, and then, submit updates as needed. Using a website requires continuous connection to the Internet.

2. Using the client locally, gives the user the opportunity to have his own personal amendments stored in his local hard drive. Moreover, these files can be shared around so that they can be verified by colleagues before being submitted to the database. This is very difficult to achieve using only web interfaces.

3. Certain processor-intensive functions, such as clustering, word alignment and computation of statistics from files can be off-loaded from the server, allowing the latter more room to service clients which are using the dictionary (see later).

4. Websites allow users to use the latest functionality exposed by the server (provided the website is updated accordingly). This can be also provided in our case by designing a client application that accepts plug-ins which can be loaded during runtime.

5. Some of the systems mentioned in Section 2.4, concentrate mainly on the linguists, thus restricting the task of building the word list (this is one of the aims of the Maltilex project (Rosner et al., 2000)) solely to them. This is due to the fact that such a system allows *direct* modification of the lexicon database. Apart from being unsafe, in the sense that a linguist might either corrupt the database, or an external user might hack an existing account, this system cannot easily be extended. By providing client tools which are publicly available, other people (including non-university staff) can contribute towards the construction of the lexicon/dictionary. This can be achieved by disallowing indirect access to database, and instead saving such amendments in an intermediary store. These can then be finally reviewed/updated by qualified people before being committed to database. A proper administrator tool is available for this purpose. To break into the database, one would have to first steal the administrator tool, and then get to know the administrator password(s) in some manner.

Although the client and administrator software perform distinct functions, a common bare bones client application supporting only basic functions can be built. This can then be later extended easily using the plug-in system. The necessary functions that are embedded into the client application include user logging in and logging out, an integrated web browser allowing users to browse the main site (see later), a simple phonetic character map, and a rudimentary real time messenger allowing users to communicate with each other. The users have also the option of signing in or out of this messenger system[3]. Having the basic client, the respective user can then obtain the necessary modules and integrate them in. That is, both client and administrator plug-ins are supported by the same piece of software. Any plug-in that specifies a pre-defined interface required by the common client application can therefore be integrated into the system without too many problems.

3.1.2 Database Design and Access

The task of designing an appropriate database structure is of paramount importance since it must complement the design discussed above. The Database Management System (DBMS) takes the role of a back-end storage, and all data traffic going in or out from clients must in one way or another pass through it. To provide a clean

[3]Although this is the basic client system, one cannot exclude that future updates may involve modification of this bare bones client.

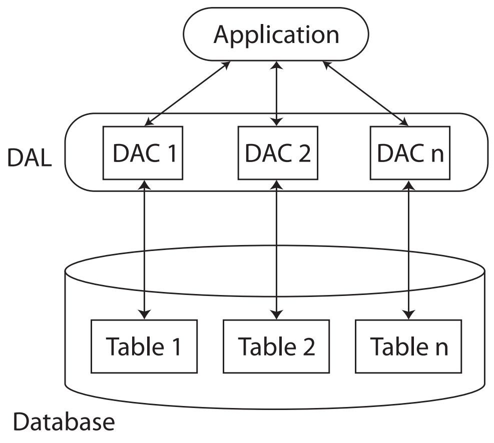


Figure 3.2: All data flow from/to the database must pass through the DAL. This provides a standard way of accessing the data.

adapter between the application programs and the actual data store, an extra layer, which we will refer to as the Data Access Layer (DAL), was introduced. The function of the DAL is to provide a clean programming interface to the applications making use of it. For added performance, some of the DAL components make use of the ADO.NET model, allowing disconnected editing of the data, as well as batch updates and insertions. Moreover, ADO.NET provides support for reading and writing database records to XML files, a feature which is indispensable in allowing us to easily save user amendments on the hard drive.

In designing the DAL components (DAC), a modular approach was taken. We required that any given table in the database, is equipped with its own DAC, and therefore, a suitable mapping from programming language constructs to database records is catered for by the DAC. To further make the DAC as general as possible, all data input is fed to the DAC in *string* format, and then, it is up to the DAC in question to identify the correct database types and convert the data appropriately. In this way, semi-independent database access was achieved[4] (Fig. 3.2).

Care must be taken so that the appropriate tables are structured sensibly, and that all the necessary relations and restrictions are set. The technique used to build the database was to place the most basic data items (in this case words) at the bottom of the data hierarchy and then, augmenting this data by adding further layers (tables),

[4]In some cases this was impossible to achieve since data conversion, say from bytes to Unicode, was needed.

all depending on this data. In this way, if this basic data is changed or deleted, these changes are propagated up this data hierarchy. More specifically, if the basic data structure is just a word list, this can be augmented by say, a dictionary table allowing users to add meaning or part of speech categories to words. This was done in order to exploit the DBMS features like relationships and constraints, thus freeing our code from such responsibilities. Moreover, custom made user queries involving table joins can be made so that specific data is easily extracted from different tables.

3.1.3 Dynamic Link Library System

One of the advantages C# offers over other languages is the easy creation of DLLs and user controls which can be easily searched during runtime, and dynamically loaded and executed by the calling application. The plug-in system is built entirely on this concept, allowing for the easy addition of functionality by simply dragging and dropping DLLs into the appropriate application plug-in directory. Unlike other languages such as C or C++ for Windows 32, the .NET framework allows us to simply copy files to and from directories, and therefore, upon addition of DLLs, no registration with the system registry (unlike COM and DCOM components) is necessary.

The system has to provide a set of APIs that the programmer can use in order to programmatically access the data contained in the lexicon. By using DLLs, we extended this requirement by not only providing components allowing access to the database (i.e. DAL components) but by also providing other handy components that were used in this project, which can be downloaded for free and incorporated into other projects. Such components include User Interface (UI) controls, data structures such as tries and general tools such as lexical analyzers and standard alignment and distance calculation algorithms[5].

Web services could have been used to provide communication with the server, but because of extra layers of indirection (SOAP wrapping, etc.), we preferred to directly connect to the database server to enhance application response times as well as efficiency. Moreover, since web services do not directly support encryption, setting up an encrypted connection could involve additional work from the programmer's side. Once again, the disadvantage of web services is that they have to live on the server, and therefore, processor hungry algorithms may slow the server down. Nonetheless,

[5]The idea here is to promote component re-usability as well as the possibility of open source development.

the way DLLs were developed allow for the easy creation of web-services, since these have to be merely wrapped around. Delegating work into various DLLs allowed us also to easily create a website supporting other features. The website is discussed in the next chapter.

3.2 Clustering

One of the main features in the system is the clustering tool that allows a user to cluster a given text file of Maltese words. Unlike Dalli (2002), the clustering operation is not carried out on the lexicon server itself, but rather on the client machine. As a consequence, clusters created by the algorithm are not added automatically to the database, but must be submitted manually by the user. This is due to the fact that the clustering function is not only limited to linguists having administrator's access to the database, but to any user having a copy of the program. Thus, all clustering updates are sent to the intermediate store on the lexicon server, where these can be later reviewed by qualified people before being accepted into the database. Although the advantage of Dalli's system is that the clustering is performed automatically, where the linguist is able to fine tune clusters as needed, the advantages related to our system are:

1. Any user can participate in the clustering process, thus anyone has the chance of creating, deleting or updating clusters. Of course, the final prerogative to commit data to the lexicon belongs to the administrator(s) in charge.

2. Cluster files can be moved around, or checked by more than one user, since these are first saved on the client's machine. After reviewing, these can be submitted for consideration.

3. Clustering functions take place on client side, not on the server side, therefore off-loading work from the server.

The clustering algorithm took quite some time to develop, since not much literature was found on the subject. Of course, several papers on morphology and clustering have been considered. However, most of them treated 'stem + suffix'-based languages, which are simpler than root based languages.

"...Arabic morphology is excruciatingly complex..." (de Roeck and Al-Fares, 2000)

Arabic is of course a Semitic language, based on roots. In our opinion, Maltese would prove even more difficult to cluster, for the simple reason that it is an amalgam of *both* Semitic and Romance languages. While the general rule in Semitic languages is to give high attention to consonants, namely because only roots are preserved during morphology (which is not the case for weak verbs), in Romance languages, the vowels have to be considered the *same* as consonants, since these form part of the stem. This conflicting definition will have to do for Maltese.

3.2.1 Methods of Clustering

This subsection discusses several approaches that were taken in order to tackle the clustering problem[6]. The first three methods fail, but for an interesting reason. These failures in turn help us build the ideas to come up with an algorithm which gave the best results out of the three discussed previously.

Levenshtein Distance with Affix Stripping

As already seen in Subsection 2.3.2 on page 21, using Levenshtein distance alone is not sufficient, and at least superfluous affixes must be stripped from words. We are using the term 'stripped' on purpose — to be able to stem a given word correctly, a considerable amount of knowledge about the morphology of the word must be known by the program. A method where consonant-vowel (CV) templates are used to detect headwords by literally reverse engineering the morphology of a word in a specific word form, is discussed in Rosner et al. (2000). But analyzing the morphology of words to that extent was not an original part of the project, so we had to use more simple techniques, such as those presented in Déjean (1998).

Definition 3.1 (Stemming) For our purpose, the stemming process is the removal of any known prefixes or suffixes from a given word. The result of this stemming process is a substring, supposedly free from its affixes. Stemming is performed using the longest match algorithm, that is, taking the longest affix possible, and trying to remove it from the word under analysis. Because we are dealing with roots, and the

[6]In this project, words are clustered according to their orthographic form.

majority of Maltese words originating from Semitic have three root consonants, the stemming algorithm requires that its output word contains *at least* three consonants[7]. Also, stemming removes adjacent double letters and properly encodes the Maltese letters 'ie' and 'għ' in order to prevent them from being split. The apostrophe (') at the end of the word is also encoded into an 'għ' to ensure comparability with words sharing the same root[8].

Because of the problem of infixes (which are not present in 'stem + suffix'-based languages), Levenshtein distance coupled with stemming is simply not enough for verbs originating from Semitic. An improvement would be to use weighted (or generalized) Levenshtein distance where weights for vowels and weak letters[9] are low, and weights for consonants are high. Although this will partially work to some extent, the following problems occur:

- Stemming must be performed accurately, so as not to leave parts of affixes which can increase the distance between two strings. Also, we cannot afford to lose any root consonants by incorrect stemming. Since we are using a set of affixes over the language (see Subsection 2.2.1 on page 14), we cannot pretend this kind of precision.

- If the morphology is relatively complex (as is the case for the Semitic part of Maltese), the distance between related words is large. Although it gives satisfactory results for Romance words, this distance function is not applicable to Semitic words.

A further augmentation to this idea would be to start off with a set of weights, train these in some appropriate manner, and then, use these as weights with the Levenshtein distance function. The idea here is to train the algorithm on a set of manually clustered items, and optimize the weights for that training set. By examining the trace-back results obtained from the distance table used during Levenshtein distance

[7]In our algorithm, we did not adopt a recursive morphology like Goldsmith did. This is because our affixes are found for a language, rather than on a word by word basis. If prefixes and suffixes are recursively defined, we would be unable to decide in which order they should be applied in order to form a proper word in the language.

[8]It is not always the case that an apostrophe at the end of a word signifies a hidden 'għ'. However, the number of such cases is very small, as opposed to the former case.

[9]There are six vowels in Maltese: 'a', 'i', 'u', 'e', 'o' and 'ie', while the weak consonants are 'j' and 'w'. Note that a weak consonant can very much be part of the root. However, there is no way to identify such cases, except by using a dictionary.

computation, an idea of which letters are being substituted can be formed, and a reduction of their weight value can be done so that for example, certain infix patterns can be learned. The problem with this method is that once weights are set, they are fixed and must be used for new words to clustered. Now, on the occasion that enough data is contained in the training set, consonants in the matrix will lose their weight, and gradually, the more the algorithm trains, the more the weight of consonants reaches the weight of vowels. Clearly, this is undesirable for words originating from Semitic.

Bi-gram Similarity

The clustering method presented in de Roeck and Al-Fares (2000) was tried on a set of Maltese words, and proved to be more effective than the methods discussed above. This is due to the fact that apart from performing light stemming, the algorithm takes into account possible infixes, and concentrates mainly on the root consonants, rather than on vowels. Because Semitic (stemmed) words, are relatively short when compared to Romance words,

> "Bi-grams with single character overlap and blank insertion (*) at word boundaries raised the SC for words sharing a root..." (de Roeck and Al-Fares, 2000)

This setting was used as a starting point for our experiments and evaluation of whether this method was suitable for Maltese or not (see Subsection 2.3.2 on page 22). However, the following modifications where applied:

1. Bi-grams containing either weak consonants or vowels were given a weight of 0.75. This is higher than the weights for weak letters used by the original algorithm, due to the fact that we are not dealing with only Semitic words (concentrating on roots) but also on Romance words (concentrating on stem).

2. Although the original algorithm uses the formula $SC =$ (shared unique bi-grams) / (sum of unique bi-grams in each string - shared unique bi-grams) to reduce the relative impact of unique shared substrings due to the very short length of Arabic words, we used the equation $SC =$ (2 $\times$ number of unique bi-grams) / (sum of unique bi-grams in each string) because once again, we have to deal with stems as well.

Clustering was done using the average-linkage algorithm, which however was modified so that clusters remained separated (i.e. not all the dendrogram is built). It was required that all elements in a given cluster had a similarity greater than some constant α. During experimentation, varying the value of α had the following consequences for our test set:

- When α was high (> 0.5), the number of clusters increased, and clusters had relatively very few elements in them. Of course, the clustering precision was high in the sense that almost all clusters contained elements which were correctly classified. Much of the clusters had only one element in them.
- When α was low (< 0.5), the number of clusters decreased, and clusters had a relatively high number of elements. The clustering precision was reduced greatly.
- When $\alpha = 0.5$, there was still a high number of clusters compared to the manually clustered items. Precision was mediocre, but it was the best out of all three tests.

Upon examining the clustering results, an anomaly was immediately detected: Maltese words sharing the majority of the root consonants were being clustered together. However, after further examination, it was found out that the reasons for such results were devoted to:

Incorrect stemming of words Since affixes are specified over a language, we cannot always expect that accurate results are produced. Therefore, in some cases, a part of the affix may be left attached to a word, or else, a root consonant or part of the stem is removed by incorrect stemming. This will have undesired effects later during clustering.

Different word forms for the same root Some of the Semitic words have highly inflected word forms, involving the introduction of additional consonants between the consonants of the root. In some cases, the resultant words are so diverse that the algorithm fails to relate them. Table 3.1 illustrates this argument.

Trying to solve this problem by varying α is quite impossible because whatever this value is, words which are totally different but have close similarity will still get

Word	**Stemmed Word**	**Unique Bi-grams**	**Score**
marad	*marad*	*m ma ar ra ad d* mr rd = 8	7
mrajjed	*mrajed*	*m mr ra aj je ed d* rj ae jd = 10	8.5
Shared unique bi-grams		*m mr ra d* = 4	3.75
$SC = (2 \times 3.75)/(7 + 8.5) = 7.5/15.5 = 0.484$			

Word	**Stemmed Word**	**Unique Bi-grams**	**Score**
marad	*marad*	*m ma ar ra ad d* mr rd = 8	7
qarad	*qarad*	*q qa ar ra ad d* qr rd = 10	7
Shared unique bi-grams		ar ra ad d* rd = 5	4.25
$SC = (2 \times 4.25)/(7 + 7) = 8.5/14 = 0.607$			

Table 3.1: As can be seen from the two computations, although 'marad' (he became sick) and 'mrajjed' (a small sick person) are related, their score is very low when compared to 'marad' and 'qarad' (he removed the stain off the clothes by rubbing). Note that although between the latter two words, there is a difference of one consonant, the root *is* different, and therefore, these should cluster separately.

attracted towards a cluster faster than those words having a same root pattern but a more complex morphology. This effect may be even worse (as in our case) if the stemmer generates wrong segmentations.

Clustering Based on Root Consonants

Building on the method of clustering discussed above, we will try to tackle the problem of clustering based primarily on the root consonants. Although this method will fail in some cases, as seen later, it provided the best results for our test set. This test data can be found in Appendix A on page 78.

We now start by trying to identify a solution for the two problems encountered by the algorithm discussed above. The problem of stemming, although cannot be solved completely, can be significantly reduced by a rather simple solution — not stemming at all. The second problem can be solved by considering only the consonants of a word for the time being. Considering the root consonants, and forming initial sets of words sharing the same root consonants can be easily implemented thanks to a standard regular expression, one for each cluster. This regular expression (which in reality is a 'personalized' similarity function for a given cluster) will allow us to accept or reject items in a cluster. The first step of the algorithm thus follows:

1. Sort the list of words to be clustered in ascending order, according to their length l.

2. For each word, form a cluster containing only that word. While forming the cluster, set the cluster centroid to the value of the word, and also build the regular expression by simply taking into account its consonants.

3. Cluster words according to the regular expression (binary similarity function).

For example, the regular expression for the word 'maqtul' would be '*m*q*t*l*'. Note that no stemming was performed. Because of the mixture of Semitic and Romance words that may be present in the list to be clustered, some Romance words may be attracted to a cluster containing its stem consonants. This may group unrelated words into the same cluster. The sorting of the word list was done so that the most general items (in regular expression sense) would be grouped at the top. These in turn will attract the majority of the less general elements to their cluster. Words with two consonants or less were not assigned a regular expression.

Because words may be incorrectly attracted to a given cluster (due to a Semitic and Romance consonant clash), a cleanup step is required. All formed clusters are scanned, and each element in the cluster is pair-wise aligned with the cluster centroid, and if the difference between these two is greater than a certain threshold (the average alignment distance in the cluster), the element in question is removed and placed into a *buffer list*. The purpose of this list is to hold all those clusters with one element that are to be clustered in the next step. Optionally, clusters containing one element are also removed from the original list and placed into the buffer list.

Calculation of distance between elements (after optionally stemming them) in a cluster and its respective centroid using global pair-wise alignment is done as follows (note that this distance function does not obey the metric axioms). First, the two words are aligned, using the weights $\sigma(a_i, -) = \sigma(-, b_j) = -2$, $\sigma(a_i, b_j) = 2$ for a match and $\sigma(a_i, b_j) = -1$ for a mismatch. Once the words are aligned, the following score is computed:

- If the letter is aligned with a gap, and it is a consonant, add -1.0 to the score, else if the letter is a vowel, add -0.5 to the score.

- If the aligned letters mismatch, and one of them is a vowel, add -0.5 to the score, else, add -1.0 to the score.

- If the aligned letters match, add 0.5 if both are vowels, else add 1.0 to the score.

After cleanup is performed, we remain with a set of clusters containing at least two elements, as all other clusters containing one element are put into the buffer list (if this step is performed). These clusters are an approximation to the seed points in the list, that is, the number of different roots in the word list to be clustered. Now, using the seed points together with the buffer list of one-element clusters, we can perform a clustering algorithm similar to k-means, clustering the existing elements using the similarity function in de Roeck and Al-Fares (2000), modified as described above. During this clustering step, no elements are allowed to change clusters once they have been placed. This is done so that the efforts done in the first pass are not lost, in that elements in the first pass are virtually correctly placed, and these should not be permitted to change. In order to provide the user with more control and fine tuning, a similarity threshold α can be set and is used so that we are able to control the intra-cluster similarity. Chapter 5 discusses the results of this algorithm in further detail.

Although the algorithm did well on our test set, this does not mean that it will do so on other word lists. In fact, there are some cases where the algorithm is sure to perform less good. These are presented and discussed briefly below:

Conflicting consonants As already noted, the problem of having two language systems coalesced into one makes it extremely hard to cluster words correctly. This is because Semitic word consonants can be shared by Romance word stems. Therefore one can expect to find Romance words clustered with Semitic words. Identification of Romance and Semitic words is unfeasible at this stage, and one would possibly require the use of a dictionary.

Incorrect stemming Although the stemming problem was reduced to a certain extent, it still remains a difficult problem to eliminate. Simple affix stripping using affixes over a language will sometimes not be not enough, and incorrect segmentations can occur. Moreover the clustering results will depend to some extent on whether stemming was correctly performed. (The algorithm can also be used by feeding it affixes from a simple stemming algorithm like the one discussed by Déjean (1998)).

Incorrect estimation of centroids Despite the fact that the idea of having a centroid for each root makes sense in the context of Arabic words, it is a somewhat second-rate approximation for the reason discussed in the first point. While the problem of finding seed points automatically is NP-hard, we tried the best method of approximating to a set of 'sure' seed points.

Weak Verbs These are those kinds of verbs in which the root consonants appear or disappear according to the current verb form. The classical example is 'tar' (he flew) in which the second radical consonant 'j' disappears, and one can be mislead into believing that the root of this word is $\sqrt{\text{TR}}$. However, upon examination of the noun 'tajra' (kite), we are able to deduce that the real root is $\sqrt{\text{TJR}}$, where the 'j' is a weak consonant (see Appendix B on page 87). These are very much common in Maltese, and very hard to identify, even by native speakers not familiar with the Maltese grammar. Of course, we cannot expect the clustering algorithm to perform well on these. To be able to identify the roots of such verbs, the use of a dictionary is needed.

The reason we devised this method was because the one by de Roeck and Al-Fares (2000) coupled with the very complex morphology some Maltese words can take (see Table 3.1 on page 39), gave a high number of clusters with respect to this one. Once again the problem was *not* because of the algorithm, but rather because of the language intricacies. On the other hand, our method tried to estimate the seed points, and then, cluster using these as reference. When a particular mixture of Semitic and Romance words sharing the same consonants is fed, great results cannot be expected. If however a balanced list is given, satisfactory results will be produced. This is due to the fact that our aim was to keep the number of clusters manageable. This effect can be reduced by choosing not to remove clusters containing only one word, during the algorithm clean-up step. If this clean-up step is not performed, then there will be additional centroids which will be used during the last clustering phase. In addition, by varying the SC cutoff threshold α, one can vary the numbers of clusters produced together with their quality.

On the other hand, with the bi-gram clustering algorithm, it is likely that average results will be produced at the expense of a relatively large number of clusters. The more the threshold α is increased, the more correct the clusters will be, but the more clusters will be produced. The reverse effect will occur by lowering α. Our main aim was to try and strike a balance between the two, thus obtaining a reasonable amount of clusters with the greatest amount of correctly clustered words. For convenience, the pseudo code of the complete algorithm is given in Fig. 3.3 on the next page.

Root-Based Clustering Algorithm

Step 1: Approximating centroids.

1. Sort the list of words to be clustered in ascending order, according to their length l.
2. For each word, form a cluster containing only that word. While forming the cluster, set the cluster centroid to the value of the word, and also build the regular expression by simply taking into account its consonants.
3. Cluster words according to the regular expression (binary similarity function) and place the clusters in C.

Step 2: Clean-up (before computing any alignments, words can be stripped of affixes to increase similarity).

1. Initialize buffer list B.
2. For each cluster:
   (a) Find the average intra-cluster pair-wise alignment similarity by computing the pair-wise alignment similarity between each element of the cluster and its centroid, storing the total result in d. $d/|c_i| \longrightarrow \delta$
   (b) For each word w in the cluster, compute the pair-wise alignment similarity between w and the centroid $\longrightarrow D$. If $D < \delta$, remove the word from the cluster, and create a new cluster c_{new} with this word. Place c_{new} into B.
3. Optional Step: Remove all those single-element clusters from C. Each cluster that is removed should be placed in B.

Figure 3.3: The final clustering algorithm (continues).

The variation of the parameter α in the root-based clustering algorithm provides a corrective action to the possible incorrect estimation of the initial cluster centroids. By experimenting with this value, different output clusters are obtained. This combines the powerful idea of the modified algorithm by de Roeck and Al-Fares (2000), and our root-based approach to clustering. Setting α to 0 will force the algorithm to

Step 3: Re-cluster (during re-clustering, words can be stripped of affixes to increase similarity).

1. Merge the lists C and B by clustering using the modified version of the k-means algorithm, using the bi-gram similarity function discussed above. Any two clusters can be merged if the distance between them is $\geq \alpha$. Once an item is considered:
   (a) If they were merged, remove the cluster that was merged from the buffer list.
   (b) If they were not merged (i.e. their similarity was $< \alpha$, placed the (single-word) cluster at the top of the cluster list C, and remove this cluster from the buffer list B. By placing the cluster in question into C, we have incremented the possible number of initial cluster seed points, and thus, give this cluster the chance to participate in having its own set of elements.
2. Output C.

Figure 3.3: The final clustering algorithm (continued).

only consider those seed points which were estimated in step 1. By combining the two approaches together we can reach a compromise between the number of clusters yielded, as we well as the correctness of those clusters, thus in the worst case our algorithm will perform as the bi-gram clustering algorithm.

3.2.2 Other Tools

Additional tools have been developed for supporting some of the ideas discussed above, as well as to help the linguist perform text-related algorithms directly on text files. To support the Maltese characters, these files should be encoded in UTF-8. We will now briefly discuss these functions; we will refer to them in Chapter 4.

Alignment and Distance

Both alignment and minimum edit distance (MED) algorithms were discussed in Chapter 2. Although the MED algorithm was not used in the root-based clustering algorithm, it was implemented and included in one of the common DLLs made available to the programmer. Being based on weights from distance matrices instead of using fixed weights, both algorithms allow the linguist to align words, or calculate the distance between them using either a default weight matrix, or else by loading a weight file from disk. In addition, the linguist can also modify and view the existing weights using a specialized weights viewer. The MED algorithm is also able to yield the operation list used to transform the source string into the target string.

Both MED and alignment can be computed either on pairs of words supplied by the linguist, or on whole text files, by computing alignment/MED on each word pair. Due to the large amount of data that is generated, the user is prompted for a file name where these results can be saved. In the case of MED, the application also writes the operation list next to each word pair, together with the distance between them, as specified by the weight matrix.

Affix Finding and Stemming

The affix finding algorithm provided to the linguist follows that of Zellig Harris (see Section 2.2.1 on page 14). Such affixes are not generated per word, but rather for the whole language. Instead of accepting all the affixes returned by the algorithm, an affix count threshold is applied, and only those affixes having counts above that specified threshold are returned. This threshold is the average suffix occurrence, calculated by dividing the total count of suffixes by the distinct suffix count. Originally, the algorithm was developed for suffixes, but by reversing the letters of the word, the same technique can be applied to prefix finding. To optimize the algorithm and make it space and time efficient, all the data from file is read into a suffix trie prior to processing.

The reason that we did not use Déjean's algorithm is that it generalized too much, and in a language like Maltese, where suffixes are commonly used for both Romance and Semitic words, parts of the stem or root were being included in the affix. Apart from being a good approximate, the algorithm by Harris is very fast since less processing than Déjean's algorithm is needed.

The stemming function provided here is the same one discussed in the previous section, that is, affixes are read from the database, and then, using the longest match algorithm, words are segmented into prefix + stem + suffix, if appropriate. Also, adjacent double letters are conflated into one.

Statistics From File and File Joiner

A very simple but useful function is that of examining text files and extracting the number of occurrences and percentage of words contained. The word statistics function is able to automatically filter out numbers and incorrect characters thanks to the lexical analyzer written specifically for the purpose. Note that this scanner might not be appropriate for other languages, since it extracts also the apostrophe (') and the hyphen (-) letters which are part of the Maltese orthographic system.

The file joiner tool is very handy for joining multiple text files into one, where the output file contains one word per line. This is very helpful for allowing one to create word lists from separate files. In order to provide the user with more control, the file joiner tool supports the use of replacement rules, which can be used by the file joiner algorithm to replace words or characters in the source text, outputting them as specified by these rules in the target text. This is extremely useful if for example, we convert a file from Portable Document Format (PDF) to text, and then correct misspelt words or letters by replacing them accordingly[10].

By no means we aim to reach the quality of tools used for text processing, such as *WordSmith* (Scott). However, by starting the implementation of small programs which can be later integrated into a single application, we can gradually build a satisfactory suite of tools for Maltese, and perhaps also for foreign languages. Right now, WordSmith tools does not support Unicode fully, while our program does, therefore, for the time being, our application will be more attractive for Maltese. Also, as opposed to WordSmith tools, our program is free of charge.

During the project construction, a basic tool able to search the web for files containing specified words was constructed. Since it is in its early phase, and due to the fact that it had to be rapidly developed, it makes use of the searching APIs provided by the Google search engine[11]. Google offers any subscribed developer the chance of executing 1000 queries per day — a rather reasonable amount for the fact that the service is free of charge. Unlike the WebGetter component forming part of WordSmith tools, our application is able to deal with lists of words, and also parse relevant HTML content so that is it automatically saved into text files without any HTML tags whatsoever. This makes it easier for the prospective linguist to verify whether

[10]Conversion can be either done directly from tools such as Adobe Acrobat, or other tools such as *pdftotext* for Linux. The advantage of *pdftotext* is that apart from being free of charge, it is also able to extract content even from protected files.

[11]Details on how to use these APIs can be found at `http://www.google.com/apis/`.

text files are written in an orthographically correct manner[12].

[12]In many Maltese websites, especially those written in the days where Unicode was not widely established, the modified Latin characters (ċ, ġ, ħ, and ż) are written using the standard English alphabet. These texts, although written in Maltese, are useless, and correction of such files by hand is out of question, namely because a lot of Maltese words use this modified Latin alphabet. Luckily, this trend is slowly dying out, as more local councils, church-related websites and others, are using the correct Maltese letters for providing their content. This is partly due to the promotion made by governmental websites and agencies (such as CIMU `http://www.cimu.gov.mt/`).

Chapter 4

Front End Architecture

The previous chapter discussed the core aspects of the system, namely, the general framework design, including data access, as well as basic algorithms for clustering. This chapter builds on top of the preceding one, and it deals with the interface design — not only of the User Interface (UI), but also of the Application Programming Interface (API) provided to the programmer. It also discusses in fair depth, the packages included for use by the normal user and the administrator.

4.1 Client Application and Plug-ins

Whereas functions offered to clients over the web by means of browser access tend to be universally accessible, due to the fact that HTML is platform independent in the sense that it can be rendered by any web browser, the problem with this approach is that the machine hosting[1] the lexicon services, including the database, web server, chat server, and possibly other services such as e-mail, will have additional load when clients start accessing various services such as clustering. Usually, clustering tends to be very processor intensive, and having say, fifty clients executing clustering at the same time on one machine (i.e. the server) is very undesirable.

Instead of having a thin client and fat server, we choose the other way round, so that the server can be free of all processing, except that related to database access, and additional standard services such web hosting and e-mail. All the other services

[1]Of course, services such as the database server and web server can be hosted on different machines, due to the fact that the DAL allows remote connections. However, clustering in our case, *must* be executed on a single machine, since it was not implemented in a distributed fashion. Even if it was, there is a limited number of additional machines that can be used for farming.

have been moved out of the core of the system, allowing them to be executed on the client side, in the form of a simple application which is able to connect to the database to exchange data. Being written in a .NET language, the client can run on any system supporting the .NET framework, independent of platform and hardware. The .NET framework for Windows and MONO for Linux are both freely downloadable.[2].

The implementation of the client DAL and the administrator DAL is provided into two separate DLL files, where the client part is made public. Conversely, the administrator DAL is kept private, for use by authorized staff managing the lexicon database. Similarly, a client set of plug-ins makes use of the respective DAL to provide access to clients, while the administrator set of plug-ins provides suitable database access using its DAL. Client plug-ins are made available through the site. Both client and administrator plug-ins make use of the same bare bones client application; moreover, an administrator can have both client and administrator plug-ins installed on the same client application[3].

4.1.1 Client Plug-ins

As the DAL is the basic module used to provide a clean interface to the application and application programmer, it is considered first. For the scope of client access, each DAC deals with two tables, with the exception of the user management DAC. As immediately obvious, the advantage of having a DAC accessing two tables is that the programmer using the component does not need to worry about the intricacies how much tables are being accessed, whether data is cached locally or not, and other database related details. All he is cares is that there is a set of APIs which he can utilize to change the data underneath. The DAL handles also data validation as well as automatic type conversion. Also, by using appropriate documentation tools such as NDoc[4], MSDN-style documentation can easily be generated in the form of compiled

[2]We do not exclude the fact that some of the light weight functions such as lexical analysis of text files and statistics finding will be deployed also on the server, and additionally wrapped in a web service. This can be done because DACs are implemented in such a manner allowing them to be used by any front-end, such as windows forms, web forms and even web services. Additionally, thanks to the compilation into Intermediate Language (IL), these can be invoked by any other language targeting the .NET framework, such as VB.NET.

[3]This bare bones client application is distinct from the client plug-ins. The function of the latter is to provide access to the 'normal' user (referred to as clients or standard users in this report), while the function of the former is to provide plug-in support so that both client and administrator plug-ins are made usable by hooking them up in this client application.

[4]This application can be found at `http://ndoc.sourceforge.net`.

HTML files.

The reason a DAC would make use of two tables is due to the fact that a standard user cannot write directly to the database, but on the other hand, can read from the actual lexicon database in order to be able to get the most current picture of the lexicon. For convenience, we will refer to the tables where the user is allowed to write as the *amendments* tables, and the tables where the administrator has full access, the *main* tables. Since we opted to use ADO.NET for data access, we will introduce another virtual table which we will call the *local* table. The local table is simply an XML text file, containing amendments which have not been yet sent to the database. These local records are readily accessibly by ADO.NET classes, and therefore, no XML parsing is needed. Moreover, these local files, being saved in plain UTF-8 encoded text, can be easily transferred amongst users, and readily accessible by Unicode-enabled editors such Windows Notepad.

Once the user has submitted an amendment into the respective amendments table, this cannot be undone, in a similar way that once a genome sequence has been submitted by mail, it cannot be undone. That is why the local table is useful. First of all, the user does not have to complete and amendment on the spot, but may close the application, and then continue later (auto saving allows this to be done easily), or else he can transfer files around, for further reviewing, before being able to submit a final amendment to the database. Once this amendment is submitted, it is the job of the administrator to review accept or reject it.

For this project, as well as to illustrate that the concept of plug-ins holds, four plug-ins for the standard user were created, each of which accesses two tables simultaneously to support the design pattern described above. These are complemented by the administrator plug-ins which process the data submitted by any standard user. The following were implemented accordingly:

- Word and affix management.
- Cluster management.
- Dictionary management.

Word and Affix Management

The word and affix amendment plug-in allows the user to browse words from the main dictionary table, and then, amend any selected word. Words can be searched by entering any word prefix, allowing the database to return all those words starting with that prefix. This is done so that not all words are loaded into memory at once (at most, the user can read all words starting with a specified letter). To maximize database efficiency, once a query is executed, results are stored locally on the client side, and the database connection is immediately closed and returned to the database connection pool. The user can then perform additional searches in the cached data, where it can be either viewed in grid mode or in record view mode. Once a word is selected for amendment, the amendment entry manager automatically appears. For words and affixes, an amendment can either be a request for insertion or deletion.

When an amendment is made, it is not automatically submitted to the lexicon database. However, it is automatically saved to disk (the local table), so that if further editing is needed, the client can easily select this entry from a list, and modify it as needed. To be able to send either word or affix amendments to the database, any locally stored amendment must be posted. Once this is posted, it is automatically removed from the local storage[5]. Of course, the user is allowed to create new amendments without reading from the database. In this way, the client can work on new amendments even without connection to the Internet.

Cluster Management

Since clustering is performed on the client side, it was sensible to take advantage of this fact, and allow any user in possession of the client program to contribute to the clustering function. Similar to the word/affix plug-in, the user is allowed to browse clusters in the main table, and either delete a selected cluster, or else update it by adding or removing words from the cluster in a an easy manner. The application will then compute a difference on the original and new cluster lists to find out which elements changed, and formulate an update request accordingly. Note that before accepting any cluster amendments, the client must verify that all the words in the new cluster are present in the database. Once this verification occurs, this amendment is automatically saved to disk for further editing.

[5]Note that for *any* database transaction, no intermediary means are used, in the sense that the DAC directly connects to the database to perform the necessary transactions.

Dictionary Management

The dictionary is a central part of the database structure, since it is layered upon the basic word list and the clusters table to provide data with additional structure and meaning. The dictionary is not only for the use of users seeking the meaning of a word, but also for programs that would like to inquire additional information, such as the part of speech, origin, gender, etc.

Like the previous plug-ins, the dictionary allows the user to look for words with a specified prefix or letter. Additional searches can then be carried out on the local cache, once results have been read from the database. Any word can be selected and instantly amended via the dictionary entry amendment dialog. Alternatively, new amendments can be created for words, but like the cluster amendments process, only words in the dictionary can be amended. The following information can be amended by the normal user: part of speech (POS) and related information, pronunciation in International Phonetic Alphabet (IPA) symbols, and the meaning of the word. A rudimentary character map for phonetic symbols is provided to aid the user in their entry.

The part of speech field originally was implemented as a string in the database. Later it was changed into a 64-bit integer so that it would be able to encode the various attributes of a word. Using the bit field facility of .NET, any enumeration can be easily used as a bit field, where bits in this 64-bit integer are set by ORing the different enumeration members. This feature was implemented in this way for the following reasons:

- Space in the database is saved by combining several fields of related information into one common field.
- If additional parts of speech/related information are to be added (which is very unlikely since these are virtually static for the language), minor modifications are necessary. If these were present as string fields in the database, then the structure of the database must be altered, dictating a change also in the respective DAC.
- Taking also the programmer into consideration, it is in general easier to deal with enumerations rather than with strings, first of all, because enumerations can be easily combined using logical ORs. Also, the programmer already knows

his set of options, since an enumeration typically contains a set of predefined values.

Following the word information template for Maltilex in Rosner et al. (1998), the following possible values are provided (up to 63 values can be specifed):

None This value is reserved for use by the database.

Part of speech Noun, verb, adjective, adverb, pronoun, conjunction, preposition, interjection, article.

Number Singular, plural.

Origin Romance, Semitic.

Gender Masculine, feminine, common, neuter.

Tense Present, past, future, past participle.

Others Diminutive, imperative, passive, active, intransitive, transitive, reflexive.

For the three plug-ins above, all amendments posted to the database include the ID of the user effectuating the amendment, as well as the date in which the amendment was submitted. Besides, all three functions allow the user to see all posted and un-posted amendments. In the case of posted amendments, only un-processed amendments can be seen, as those which have been processed by the administrator are removed from the amendments tables subsequent to processing.

4.1.2 Administrator Plug-ins

The administrator plug-ins complement the set of plug-ins provided to the normal user by allowing the administrator to process amendments and deciding which ones are to be committed to database and which should be discarded. In addition, all administrator plug-ins allow direct access to the main tables of the database, that is, all records can be modified directly. The four administrator components are outlined below:

Word Management

Whereas in the client plug-ins, word and affix management tools were grouped into a single plug-in, they are two separate functions from the administrator's point of view. This was done because the UI of each plug-in is already more complicated in the sense that there are more functions available, and therefore, by splitting the two functions in two separate components, a much more logical view of the data is presented.

Apart from allowing the administrator to accept or reject amendments, he is also able to add, remove and updated words in the database. Since C# supports Unicode directly, by configuring a Maltese keyboard, one can easily enter the special characters used by the Maltese language. Due to the fact that insertion of words is very common, the administrator is also allowed to insert a list of words in batch mode from a file. Note that whereas manual modification of word records will instantly commit changes to the database, when batch mode is activated, changes are not committed automatically, in order to allow the administrator to review or correct the list of words before committing it to the database. Also, during this stage, any modification (update/addition/deletion) of words is not automatically committed. However, once changes are committed, the application automatically switches back to instantaneous committing mode.

Affix Management

The affix management plug-in is similar to the word management process, and in a similar way, affixes can be added, deleted or updated. The affix addition dialog allows two kinds of affixes, namely, prefix and suffix types. These affixes will be used during stemming as well as during the clustering algorithm mentioned in the previous chapter. The administrator is also allowed to add a list of affixes from file. Once again, the affixes are not automatically committed to database; the idea is identical to the one used for adding words. By default, affixes are read with their type set to suffix, on the grounds that suffixes are more common than prefixes and that there are only a handful of prefixes in existence. Also, since usually the list of affixes to be added is short, one can easily edit the list before committing it to the database.

Cluster Management

While accepting and rejecting cluster amendments is a simple business for the administrator, the DAC underneath must check what words should be added or removed from the cluster specified in the amendment. To keep things simple, and avoid unnecessary database access, updates are not performed by calling the SQL update statement, but instead we follow the idea that an update can be carried out by deleting and inserting a record. This may appear more time consuming, but first note that before a record is deleted it is fetched in the local cache, and only if it is not found, it is then retrieved from the database. This is due to the structure of the database, where data is stored in tuples like the following:

<word, headword>

If a headword if removed or changed, additional checks must be first made to verify that database constrains are obeyed. Deleting the word and adding it afterwards proves more efficient and also avoids unnecessary code for checking from the DAC. The administrator is also allowed to insert, delete and update clusters directly from the database.

Dictionary Management

The dictionary table is very useful, and in fact, its aim is to provide words with meaning and additional information. Also, an additional field, allows the administrator to supply an audio file that will be used to provide the pronunciation of the word in question. The all-important structure to the dictionary table is provided implicitly by the clusters table. Since the keys in the clusters table are the words themselves, and these are in turn referenced by words in the dictionary table, this cluster structure is also induced in the dictionary table. This will allow us to categorize words according to the headword.

Apart from providing functions similar to the ones discussed above (amendment handling, addition, deletion, etc.), another class was specifically built so that the data in each field of the dictionary table is queried. The field that is of most interest is the field containing POS and related information. Once the dictionary is complete, it should prove very useful to other applications, such as those dealing with word sense disambiguation, and other approaches where dictionaries are needed. For the convenience of the programmer requiring such a service, this class has also been

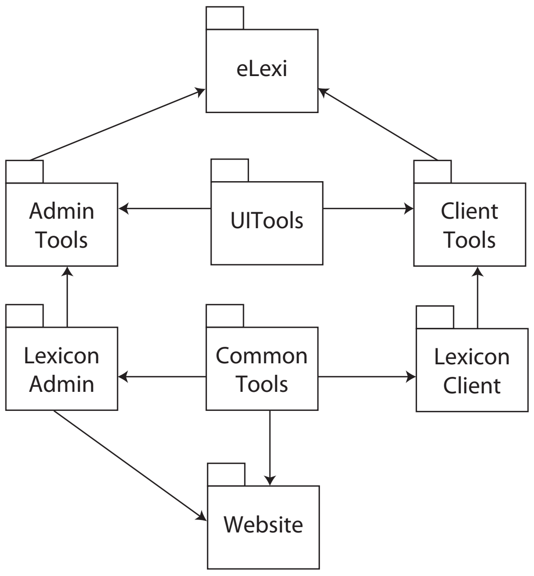
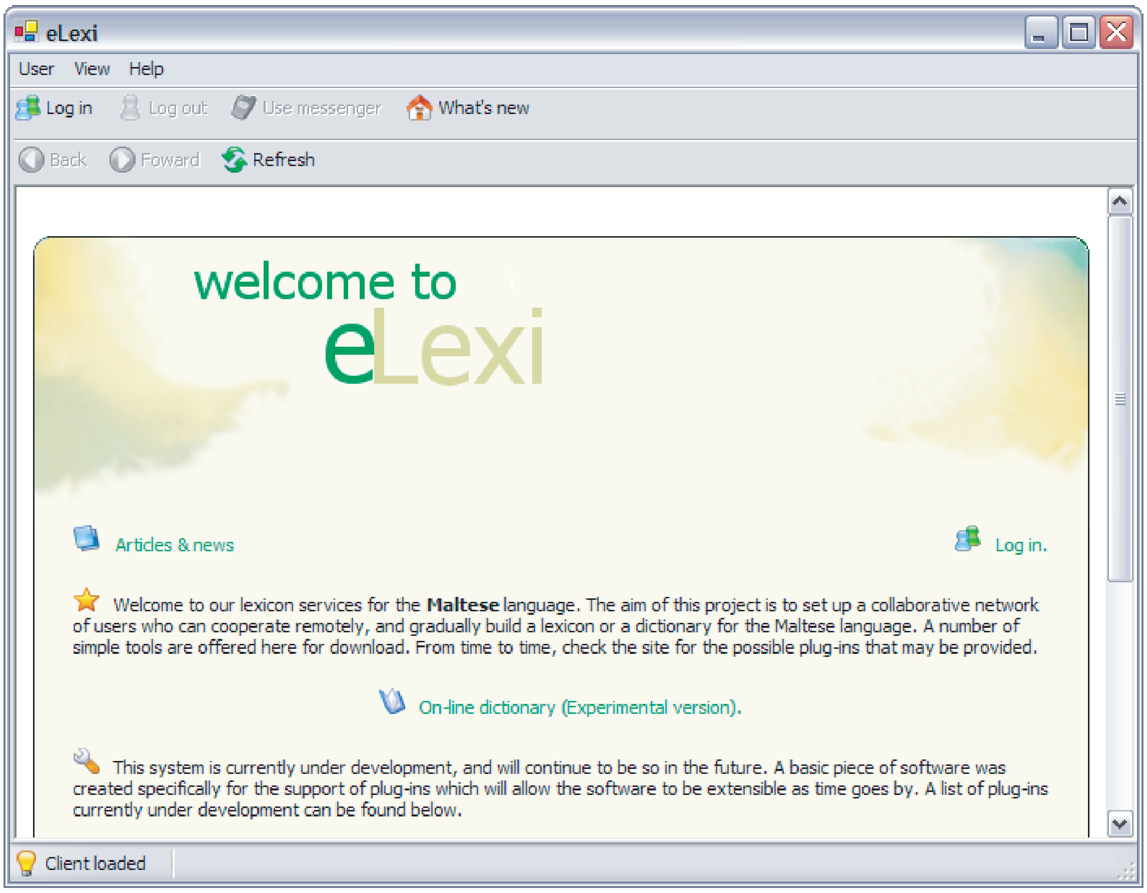


Figure 4.1: These are the components of the system, and their relation with each other. With the exception of 'eLexi', 'ClientTools', 'AdminTools' and the 'Website' components, these can be used by other programmers to aid them in the development of new plug-ins.

wrapped around in a simple web service. Fig. 4.1 illustrates the components of the system, and how they are related.

4.1.3 The Common Client Application

Without the common client application, neither the standard user nor the administrator will be able to use the functionality discussed above. This is because although both plug-ins encapsulate a lot of features, they are not able to function on their own, and must be loaded appropriately by the common application. The plug-in system was successfully implemented thanks to the use of interfaces. An interface is a contract which binds the implementer of a class with the obligation of specifying the implementation of those methods and properties declared in the interface. In our case, these obligations are very simple, and specify only the name of the plug-in, its creator, and a menu item title which is used by the common client application when displaying an appropriate title for the plug-in being used. In addition, the methods for doing any initialization and any cleanup are required, but these can be left empty if not needed. The things that will be mentioned now apply to both the standard user and the administrator plug-ins.



Welcome to our lexicon services for the **Maltese** language. The aim of this project is to set up a collaborative network of users who can cooperate remotely, and gradually build a lexicon or a dictionary for the Maltese language. A number of simple tools are offered here for download. From time to time, check the site for the possible plug-ins that may be provided.

This system is currently under development, and will continue to be so in the future. A basic piece of software was created specifically for the support of plug-ins which will allow the software to be extensible as time goes by. A list of plug-ins currently under development can be found below.

Figure 4.2: The common client application with the web browser in view.

Apart from offering a place where plug-ins can be loaded, basic built-in logging in/out of users is provided (again, logging takes place via an appropriate DAC built specifically for this task). This login information is stored globally in the application, and can be easily accessed by those plug-ins needing this information to carry out their tasks. In addition to this, a web browser control (Microsoft's ActiveX IE control) has been provided, and this allows the user to connect to the main project website (Fig. 4.2).

A rudimentary chat program is also integrated into the common client application. Its main use is to allow users and administrators to chat with each other in a real time manner, and hopefully, increase productivity. Of course much more advanced programs like MSN messenger can be used, but the advantage of our chat program is that it allows users on the same network to chat with each other. The only thing they needed in common is a chat server. The job of the chat server is to maintain a list of connected users, and as a new users connect, forward this list to them using an appropriate protocol. Therefore, if multiple chat servers are set up on different machines (only one chat server per machine), multiple 'chat rooms' can be created, although the client can only be connected to one at any one point in time. In practice, there should be only one chat server on the machine where the lexicon server is to be hosted.

The original intent of the client was to provide services to a single user, where all his files (un-posted amendments) are placed in an appropriate folder, and accessed in read/write mode during runtime. However, with some additional code, the application was modified so that it would support multiple users in the sense that as soon as a user logs in, the application will either search for his files in the appropriate folder, and load them into the application, or else, new files are created if no such files are found. Then, the user can take these files with him, and use them in another client application on another machine.

4.2 Website

In the previous section, we discussed all the features of the common client application, together with its plug-ins. We did not mention the tools used for file processing, as there were mentioned in the previous chapter; they too form part of the collection of tools provided to the user. To complement these utilities, the website was set up so that first of all, plug-ins, plug-in and common application updates was well as bug fixes are made available. Secondly, it allows new users to register and use the system. Thirdly, the dictionary is made available in read-only mode, so that it can be accessed via a web browser.

The default page allows the user to access the various links available. A page where new articles are set up, and a few links where the software can be downloaded are provided. In addition to these, new users can register with the lexicon server, and by using an appropriate DAC for user handling, the website can accept new users on demand. During the registration process, a user must enter his user name (e-mail since by default, it is unique although checks are still made. The e-mail can also be used to contact users.), and other details such as his name, surname and country. For extra security and to force the user to specify a correct e-mail, the password is not entered by the user, but it is randomly generated by the system, and mailed to the him, together with instructions on how to use the account. Once this password is known, it can be changed, together with any other details later on. These credentials are used so that the user can log in and use the application together with its plug-ins. Note that plug-ins such as those used for amendment submission make use of the user name to mark which amendment belongs to whom. Any registered user is allowed to unregister by providing his user name and password.

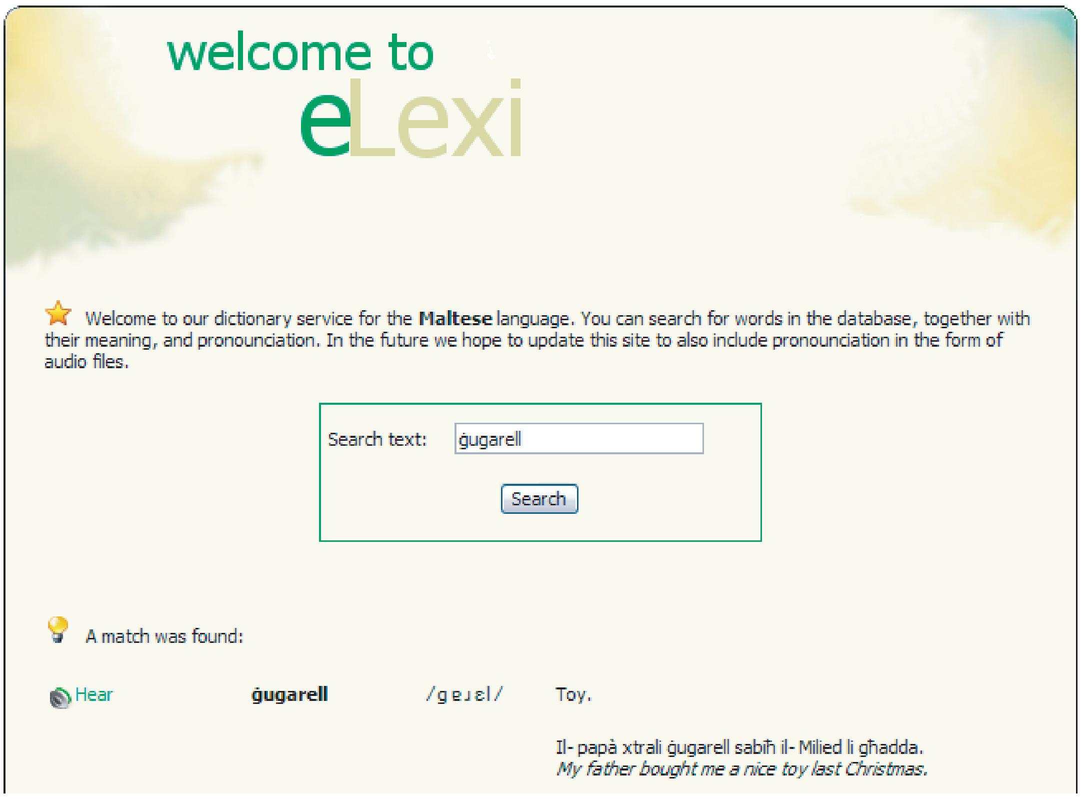
Although any user can register and download the software, only the standard user plug-ins are provided; this prevents users from having direct access to the database. Although the administrator plug-ins are kept private, they still have to make use of the common client application. This illustrates the generality of our approach, where the application is not only shared by the two parties, but it can also be extended easily through the use of plug-ins.

The dictionary page provides a very basic search facility in which information about words can be displayed in an appropriate manner[6]. If a file to the audio pronunciation exists in the respective audio path field, then an additional icon, together with an appropriate link to the file, are provided. This will allow the user to either save the file or play it using some audio player. By entering appropriate HTML tags in the meaning field (during dictionary construction), one can enhance the display of meaning for a given word or word examples.

The clusters that were constructed earlier now prove very useful in the logical structuring of the dictionary, and provide the user with the facility to look for related words, that is, words in the same cluster. In our case, this was achieved by querying the clusters table for words sharing the same headword and concisely presenting the results in a combo box allowing for easy navigation between different members of the same cluster. The ultimate aim of the dictionary is to reach a high quality standard, such as the one by Aquilina.

These days, web browser tool bars are becoming widely popular since they provide a piece of self-contained functionality which is able to offer certain services to the user without him or her navigating to the website offering the services in question. A very simple but effective dictionary tool bar has been developed, and it can be readily installed and used from within Internet Explorer, allowing the user to search for words in an easy manner (Fig. 4.4).

[6]There exists a basic English-Maltese dictionary on the web, written by Grazio Falzon. However, apart from using English letters for the special Maltese characters, this dictionary is not searchable. This dictionary can be found at: `http://aboutmalta.com/language/engmal.htm`



Welcome to our dictionary service for the **Maltese** language. You can search for words in the database, together with their meaning, and pronounciation. In the future we hope to update this site to also include pronounciation in the form of audio files.

Search text: ġugarell

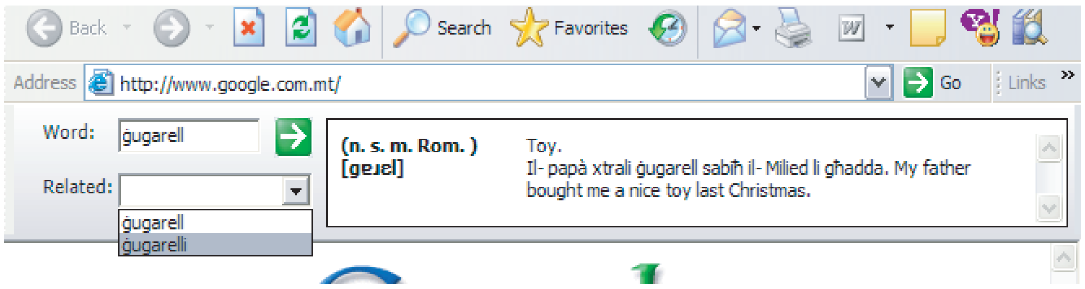
Search

A match was found:

Hear **ġugarell** /gɐɹɛl/ Toy.

Il- papà xtrali ġugarell sabiħ il- Milied li għadda.
My father bought me a nice toy last Christmas.

Figure 4.3: Searching the dictionary.



http://www.google.com.mt/

Word: ġugarell

Related: ġugarell, ġugarelli

(n. s. m. Rom.)
[geɹel]

Toy.
Il- papà xtrali ġugarell sabiħ il- Milied li għadda. My father bought me a nice toy last Christmas.

Figure 4.4: The dictionary bar.

Finally, as an open-source development effort, a very basic forum was set up so that possible future amendments, requests, and bug reports are submitted by users and discussed openly — this will enable the software to evolve in a continuous manner.

Chapter 5

Observations and Results

Having discussed the core operation of the framework and dealt with the clustering algorithm in Chapter 3, together with the user and programming interfaces in Chapter 4, we now present a series of tests that were carried out on our clustering algorithm, as well as on the modified algorithm by de Roeck and Al-Fares (2000). In addition to these tests, we shall also compare some frequency tests run on small corpora with the ones obtained by Dalli (2002) for the corpus he used[1].

5.1 Evaluating the Clustering Algorithm

In this section, we compare the two algorithms mentioned previously, and highlight their differences, strengths and weaknesses with respect to our test set. To be fair, we also tested the algorithms on an extract from the Maltese bible. A piece of text was selected specifically from the bible since biblical texts tend to contain a good mixture of Semitic words; this allows us to truly test both the algorithm by de Roeck and Al-Fares (2000) (which was specifically developed for Arabic, although it was modified as explained in Chapter 3) with our own. Since this biblical text includes weak verbs, a list of each word with the corresponding root will be provided — this will illustrate the complexity that both algorithms have to deal with.

The *objective* of our evaluation is to test that the clustering algorithms produce clusters with high correctness, while at the same time, producing a number of clusters which approaches in a satisfactory manner the original number of clusters in the test

[1]It must be stressed that some spelling mistakes are bound to be found in the texts used for these tests. No attempt to correct such mistakes were made, since this is actual scenario in which the algorithms will work.

set. In other words, we are aiming for an output containing a set of clusters which tries to identify all (or almost all) possible clusters in the original test set without being too general (placing unrelated words in the same cluster, thus reducing the number of clusters produced) or too specific (placing very closely related word variants in the same cluster, thus increasing the number of output clusters).

5.1.1 Clustering of Dictionary Text

A set of clusters adapted from the dictionary by Aquilina will be used as a reference to the tests that will follow. Although the text used here could have been taken from any other source, and then clustered manually, we chose the dictionary approach namely because of the great variation of words that can be found, and also, because clusters are sure to be correct. For our purpose, a cluster is a headword entry together with its related words. This test (which contains both Romance and Semitic words) accompanied by the list of affixes that were used for stemming[2] can be found in Appendix A on page 78.

The Modified Bi-Gram Clustering Algorithm

For the purpose of these tests, we will refer to the algorithm by de Roeck and Al-Fares (2000) as the bi-gram clustering algorithm. This algorithm, which measures the similarity between words using bi-grams, uses a threshold parameter α (called the SC cutoff) which allows words with a similarity higher than α to cluster together. The following table illustrates the results of these tests with various values for α. Our test set contained 80 clusters, that is, 1309 words. First, it was seen fit that for these tests, no weak verbs would be allowed, since the algorithm cannot handle these correctly. However, later it was decided that the introduction of just two clusters containing weak verbs will add a little noise to the test set, and therefore, we could see how both algorithms perform under these conditions, since after all, the setting in which they will work will certainly contain such verbs. Also, we used the average-linkage clustering method for both tests.

[2]We are providing this list since the result of clustering partially depends on this factor.

Value for α	**Total clusters**	**Correct clusters**	**Incorrectly clustered words**	**Singe-word clusters**
0.7	624	624	0	330
0.6	424	410	19	173
0.5	268	236	66	92
0.4	146	113	93	27

Value for α	**Average words/cluster**	**Average wrong words/cluster**
0.7	2.098	0
0.6	3.087	0.045
0.5	4.884	0.246
0.4	8.966	0.637

Before discussing the results in the tables provided above, we should state how a correct *multi-word* cluster is defined. We shall adopt the same definition used by de Roeck and Al-Fares (2000):

> “A correct multi-word cluster covers at least two words and is found in the manual benchmark. It contains all and only those words in the dataset which share the root.”

As pointed out by de Roeck and Al-Fares (2000), this is a *recall-type* definition, and it cannot directly measure the quality (precision) of a cluster. Expanding on this, an *incorrectly clustered word* is one which does not share the root consonants or stem in a given cluster. In the tests we have carried out, resultant clusters have been manually analyzed, and incorrect words were selected in such a manner so that their number is kept as low as possible. For example, if a cluster is labeled as ‘intiret’ (inherited), but the rest of the cluster contains ‘intiret’ (the cluster centroid must *still* be present as a cluster element), ‘inġabret’ (she was picked up/she was collected/she improved herself), and ‘ġabriet’ (collections), then ‘intiret’ is marked as the incorrectly clustered word. This is because, the cluster headword is chosen by simply finding a word in the cluster which is closest the all other words in terms of similarity.

Examining the tables above, it can be seen that when the SC cutoff value α is very high (0.7), there are no incorrect clusters generated. However, the number of clusters is overwhelming with an increase of 544 clusters, out of which 75% are single-word clusters. Decreasing α to 0.6 does not improve the situtation very much, and still, a high number of clusters is generated. Even though the number of clusters is high, we can already see words which are incorrectly clustered — this highlights the fact the Maltese morphology so intricate that some words are mistaken for others, and are attracted to a specified cluster centroid before other variants of the centroid in question.

Setting α to 0.5 and 0.4 respectively reduces the number of clusters created at the expense of an increase in the number of incorrect clusters. While with α set to 0.5, incorrect words in clusters mostly amount to one or two, with an SC cutoff value of 0.4 the number of incorrect words in some clusters increased to higher values[3].

The Root-Based Clustering Algorithm

By measuring similarity using bi-grams, the bi-gram clustering algorithm is able to cluster words based on their common set of bi-grams. Although the results generated by this approach were good, when compared to other known distance/similarity measurement algorithms, one immediate downside, as seen from the tables above, is the amount of clusters that is generated. One cannot simply draw a straight line (or plane) in order to separate a set of words into different partitions; varying α will either produce a manageable number of clusters and in some way or another cluster unrelated words into the same cluster (if α is low), or else, produce a high number of clusters with correctly clustered words (if α is high). Our *aim* is to try and reduce the number of clusters, preferably as close as possible to the original number (80), while at the same time, keeping the number of incorrect clusters to a minimum.

The root-based clustering algorithm (see Section 3.2.1 on page 39) was executed on the same test set used for the algorithm discussed above. The results of this test are displayed below:

Value for α	**Total clusters**	**Correct clusters**	**Incorrectly clustered words**	**Singe-word clusters**
0.0	77	55	70	0
0.1	77	55	70	0
0.2	80	60	63	1
0.3	101	90	35	8

[3]In one case, 14 words were incorrectly placed in a cluster containing 31 words.

Value for α	Average words/cluster	Average wrong words/cluster
0.0	17	0.909
0.1	17	0.909
0.2	16.363	0.788
0.3	12.960	0.347

As can be seen from the results above, our algorithm produces a manageable number of clusters which however contain few errors. Setting the value of α to 0 will force the algorithm to use only its initial estimated seed points. Since we have a lot of variants for the same stem or root, we expect that this algorithm performs better when compared to the bi-gram clustering algorithm. When slowly increasing α to 0.3, we notice that the number of single-word clusters increases, while the number of incorrectly clustered words decreases; the price to achieve these values is paid by the relatively small increase in the number of clusters.

It must be noted that in both algorithms, there will almost always be incorrectly clustered words. In particular, for our algorithm, even though we first filter words according to the consonant pattern, we still get incorrectly clustered words. Once again, this is due to the complex morphology of the Maltese language, where some words are so 'mutated' by the morphological processes that they are unrecognizable from their root. Due to their stem model, this phenomenon is rather uncommon for Romance words.

5.1.2 Clustering of Biblical Text

We now test both algorithms on an extract from the Maltese bible. As in the tests above, for the bi-gram clustering we modified the value of α from 0.7 down to 0.4 in steps of 0.1 intervals. Note that while the dictionary test contained unique words, this text is bound to have repetitions. To make the test as fair as possible, proper names were removed from the extract. The results are tabulated below:

Value for α	**Total clusters**	**Correct clusters**	**Incorrectly clustered words**	**Singe-word clusters**
0.7	108	107	1	67
0.6	98	96	3	55
0.5	83	73	14	37
0.4	74	60	20	27

Value for α	**Average words/cluster**	**Average wrong words/cluster**
0.7	2	0.009
0.6	2.204	0.031
0.5	2.602	0.169
0.4	2.919	0.270

At a glance, a lot of incorrectly clustered words, with a high number of clusters appears, no matter what the value of α is, and to make things worse, the more α is decreased, the more the number of errors increases. But before drawing any conclusions, consider the results of the root-based clustering algorithm:

Value for α	**Total clusters**	**Correct clusters**	**Incorrectly clustered words**	**Singe-word clusters**
0.1	18	1	110	1
0.2	34	11	87	5
0.3	51	29	44	13
0.4	74	60	21	27

Value for α	**Average words/cluster**	**Average wrong words/cluster**
0.1	12	6.111
0.2	6.353	2.559
0.3	4.235	0.863
0.4	2.919	0.284

The result which immediately draws our attention is the one in which $\alpha = 0.4$ for the bi-gram clustering algorithm, and $\alpha = 0.4$ for the root-based clustering algorithm, where the results are virtually identical. This is one feature of the root-based clustering algorithm, where in its worst case it virtually behaves like the bi-gram clustering algorithm. This was possible by the introduction of the α parameter, which allows the root-based clustering algorithm to be able to generate single-word clusters when not enough evidence of similarity is available. Like in the bi-gram clustering algorithm, the SC cutoff allows single word clusters not to be combined with others, if their similarity is not greater than or equal to α. If α was not introduced, the modified k-means algorithm used to perform the final clustering (step 3) would have been forced to place *all* single-word clusters into any cluster, according to their similarity. Therefore, if there exists no similarity between two given clusters, these will *still* have to be clustered together, since the logic of the standard k-means algorithm is to place all input objects into the specified seed points. Loosening this restriction by introducing α allowed the algorithm to cope with single-word clusters.

Now, if we consider the results displayed in the tables above, one may conclude that *both* algorithms performed badly when compared to the clustering of the dictionary text. However this is not precisely the case, namely because:

Unrelated words Usually, clustering is successful when it is applied on a set containing items related in some manner, even though these relations are not always obvious. In the case of this particular bible extract, words were very unrelated, and therefore, the creation of single-word clusters was inevitable.

Lots of articles While in the dictionary text there was no presence of articles, in this text (and in the majority of other texts in Maltese), the presence of articles is very prevalent. Apart from this, the basic form of the article varies according to the first consonant of the word following the article[4]. In this test, most of the time, articles (i.e. words ending in '-') were clustered together.

Missing affixes The affix list used in the clustering test performed over the dictionary text is of course not exhaustive, and it was specifically built just for the dictionary test. Therefore, some affixes may be missing, leading to a possible incorrect clustering of certain words. Although this effect is minimal, it should

[4]E.g. il- qamar (the moon), ix- xemx (the sun), iż- żiemel (the horse). As opposed to English, the article varies according to the word following it, while in the case of English, the same article 'the' is used; in German, only three articles are used, namely 'der' (for masculine), 'die' (for feminine) and 'das' (for neuter). To complicate things further, when the word following an article denotes something which is countable and singular, the hyphen precedes the article, as shown: kemm -il darba (on a regular basis/often), ħdax -il plejer (eleven players). See Appendix B on page 87 for more details on the article.

be made explicit that ultimately, the presence/absence of affixes will in some way affect the results of the clustering operation.

Weak verbs It was already made clear that weak verbs will confuse the clustering algorithm since at times root consonants are visible in some verb form, and hidden in others. Consequently, we do not expect that these are correctly clustered, unless weak verbs are identified beforehand. Needless to say, weak verbs are very common in the Maltese language and are widely found in virtually any Maltese text. The table below illustrates all the verbs in the extract taken from the bible, where these are accompanied by their root if appropriate.

Word	Stem/Root	Word	Stem/Root	Word	Stem/Root
f	—	inkun	$\sqrt{\text{KWN}}$	tagħarfuh	$\sqrt{\text{GħRF}}$
dak	—	jien	—	rajtuh	$\sqrt{\text{R(')J}}$
iż-	—	tkunu	$\sqrt{\text{KWN}}$	urina	$\sqrt{\text{WRJ}}$
żmien	$\sqrt{\text{ŻMN}}$	intom	—	jkun	$\sqrt{\text{KWN}}$
qal	$\sqrt{\text{QWL}}$	ukoll	—	biżżejjed	$\sqrt{\text{ŻJD}}$
lid-	—	t-	—	għalina	$\sqrt{\text{GħLJ}}$
dixxipli	dixxipl	triq	$\sqrt{\text{TRQ}}$	ili	—
tiegħu	$\sqrt{\text{TGħ}}$	għall-	$\sqrt{\text{GħLJ}}$	daqshekk	$\sqrt{\text{DQS}}$
tħallux	$\sqrt{\text{HLJ}}$	tafuha	$\sqrt{\text{GħRF}}$	magħkom	$\sqrt{\text{MGħ}}$
qalbkom	$\sqrt{\text{QLB}}$	qallu	$\sqrt{\text{QWL}}$	int	—
titħawwad	$\sqrt{\text{HWD}}$	aħna	—	għadek	$\sqrt{\text{GħWD}}$
emmnu	$\sqrt{\text{WMN}}$	nafux	—	għaraftnix	$\sqrt{\text{GħRF}}$
alla	—	inti	-	ra	$\sqrt{\text{R(')J}}$
u	—	kif	$\sqrt{\text{KJF}}$	lill-	—
fija	—	nistgħu	$\sqrt{\text{STGħ}}$	tasal	$\sqrt{\text{WSL}}$
wkoll	—	nafu	—	tgħid	$\sqrt{\text{GħJD}}$
fid-	—	wieġbu	$\sqrt{\text{WĠB}}$	temminx	$\sqrt{\text{WMN}}$
dar	$\sqrt{\text{DJR}}$	jiena	—	fil-	—
ta	$\sqrt{\text{GħTJ}}$	hu	—	kliem	$\sqrt{\text{KLM}}$
missieri	missier	is-	—	ngħidux	$\sqrt{\text{GħJD}}$
hemm	$\sqrt{\text{HMM}}$	sewwa	$\sqrt{\text{SWJ}}$	rajja	$\sqrt{\text{R(')J}}$

Word	Stem/Root	Word	Stem/Root	Word	Stem/Root
ħafna	$\sqrt{\text{HFN}}$	l-	—	iżda	—
għamajjar	$\sqrt{\text{GħMR}}$	ħajja	$\sqrt{\text{HJJ}}$	jgħammar	$\sqrt{\text{GħMR}}$
li	—	ħadd	$\sqrt{\text{WHD}}$	qiegħed	$\sqrt{\text{QGħD}}$
ma	$\sqrt{\text{MJ}}$	jmur	$\sqrt{\text{MWR}}$	iwettaq	$\sqrt{\text{WTQ}}$
kienx	$\sqrt{\text{KWN}}$	għand	$\sqrt{\text{GħND}}$	għemil	$\sqrt{\text{GħML}}$
hekk	$\sqrt{\text{HKK}}$	il-	—	emmnuni	$\sqrt{\text{WMN}}$
kont	$\sqrt{\text{KWN}}$	missier	missier	għal	—
ngħidilkom	$\sqrt{\text{GħJD}}$	jekk	—	ħaġa	$\sqrt{\text{HWĠ}}$
sejjer	$\sqrt{\text{SJR}}$	mhux	—	oħra	—
inħejjilkom	$\sqrt{\text{HJJ}}$	bija	—	emmnuh	$\sqrt{\text{WMN}}$
fejn	—	kieku	—	minħabba	—
toqogħdu	$\sqrt{\text{QGħD}}$	għaraftu	$\sqrt{\text{GħRF}}$	fl-	—
meta	$\sqrt{\text{MTJ}}$	lili	—	stess	—
mmur	$\sqrt{\text{MWR}}$	kontu	$\sqrt{\text{KWN}}$	fis-	—
nħejjilkom	$\sqrt{\text{HJJ}}$	tagħarfu	$\sqrt{\text{GħRF}}$	jemmen	$\sqrt{\text{WMN}}$
post	post	il	—	għad	$\sqrt{\text{GħJD}}$
nerġa	$\sqrt{\text{RĠGħ}}$	minn	—	jagħmel	—
niġi	$\sqrt{\text{ĠJJ}}$	min	—	nagħmel	$\sqrt{\text{GħML}}$
biex	—	issa	—	akbar	$\sqrt{\text{KBR}}$
neħodkom	$\sqrt{\text{(')HD}}$	l	—	minnu	—
miegħi	—	quddiem	$\sqrt{\text{QDM}}$	għax	$\sqrt{\text{GħX}}$

The strength of the root-based clustering algorithm is therefore shown when a lot of variants of the same stem/root are present in the set of words to be clustered, and it is hoped that if a large corpus is used, the results of the clustering operation are quite satisfactory. Yet, the clustering of small files containing a couple of hundred words (the bible extract contained 216 words) must not be taken lightly, and suitable measures were taken so that when the data to be clustered is sparse (in the sense of a high number of unrelated stems/roots), at least average results are produced. This was done by the introduction of the α parameter which allows the algorithm to behave like the bi-gram clustering algorithm, should the situation arise.

5.1.3 Word Statistics

In this subsection, we use our statistics tool to try and obtain frequencies of Maltese words and compare them to the results obtained by Dalli (2002). The following table shows the word frequencies for a very small corpus (10,276 words) taken from various Maltese websites. The topmost twelve ranking words are displayed here.

Rank	**Word**	**Frequency**	**Rank**	**Word**	**Frequency**
1	l-	5,986	7	mill-	824
2	ta'	3,901	8	fil-	817
3	li	3,578	9	fl-	693
4	u	3,005	10	dan	657
5	il-	2,831	11	f'	608
6	tal-	2,134	12	għall	591

The fact that the topmost ranks are occupied by the articles immediately suggests that these are the most common words, at least in our corpus. As already noted, articles are usually variants of the same basic article, which however changes according to the word following it. The next test is performed on a bible-related text of 38,729 words.

Rank	**Word**	**Frequency**	**Rank**	**Word**	**Frequency**
1	li	45,705	7	alla	12,795
2	l-	43,079	8	tal-	11,755
3	u	38,392	9	f	10,274
4	ta	31,539	10	dan	9,223
5	il-	28,264	11	hu	8,834
6	ma	17,303	12	kristu	7,298

Again, the articles occupy some of the higher ranks, although in this test, we have an increase in words which are not classified as articles. Particularly note the words 'alla' (god), and 'kristu' (christ) which have displaced the higher raking words seen in the first test into a lower level. By measuring the frequency of certain words in corpora, we may get a very rough indicator as to which kinds of texts were used to build the corpus. WordSmith tools uses a similar approach to find possible keywords (those words which have a relatively higher occurrence count with respect to other words) in texts.

The table of frequencies given below was taken from Dalli (2002, pg. 185). As it can be noted, even though the corpus he used amounted to about 2.39 million words, very similar frequencies were obtained, and in some cases, words even kept their rank intact. This may hint that the typicality of certain Maltese words (such as articles for example) can be determined from even relatively small collections of text. Needless to say, the more words present, the more accurate our frequency distribution will be, but in cases were rough estimates are needed, small corpora can suggest fairly good figures.

Rank	Word	Frequency	Rank	Word	Frequency
1	li	90,166	7	kien	13,811
2	l-	84,344	8	biex	13,431
3	il-	53,834	9	fl-	11,218
4	fil-	18,727	10	dan	11,000
5	f’	17,958	11	b’	10,402
6	ma	16,098	12	għal	9,583

Results

For the dictionary test, the root-based clustering algorithm outperformed the bi-gram clustering algorithm since the former approached the original number of clusters while at the same time maintained a higher cluster correctness than the latter algorithm. When both algorithms were presented with a random test set (the bible text) which contained a high amount of unrelated words together with a large number of irregular and weak verbs, both algorithms produced virtually identical results, indicating that the root-based clustering algorithm will at worst produce average results. It must be noted that irregular and weak verbs are very hard to cluster since usually their root changes between one verb form and the next. Knowledge-based techniques can be used to deal with such verbs.

Chapter 6

Further Work and Conclusions

This report detailed the work done on the thesis, and presented its main points. Its aim was to design and implement a first cooperative framework for linguists, which enables them to build a computational lexicon for Maltese. One such product of this computational lexicon is an online dictionary, providing users with searching facilities, aiming to reach high quality work such as the dictionary by Joseph Aquilina.

One way to achieve this is to decentralize the process of dictionary development. Like open source software products, where these are developed by a number of people across the globe, so can an 'open source' dictionary be developed, by not only involving linguists and academic staff, but also common people. However, control is provided to specialized people, and therefore, the final decision of admitting a word or definition into the database is theirs.

Apart from building a huge word list with no structure at all, a second aim of the project was to provide the lexicon database with structure, similar to that found in dictionaries. Using well known clustering techniques, as well as our cooperative framework of building clusters, semi-automated clustering of words directly from text files was provided. The clustering algorithm performs best when a sufficiently large text file is provided, as well as when different forms of the same stem/root are found. In the case where the data is sparse, the algorithm is still able to cope in a satisfactory manner, achieving results similar to those by de Roeck and Al-Fares (2000). Clusters generated by this algorithm can then be submitted to the lexicon database for reviewing. The advantage of an open source approach to dictionary/lexicon building is that users can review and also edit each other's work. Thus for example, a given cluster can be built by the aid of the clustering algorithm, as well as a number of users, each editing different entries in the same cluster. Although this can be done in another system we have discussed (Dalli, 2002), it is rather cumbersome, since it does not provide this distributed facility present in our system. On the other hand, one downside of the distributed approach is that words are not automatically clustered and added to the lexicon database, although the latter matter discussed in Dalli (2002) still involves a post editing phase.

This distributed framework is supported mainly by the design of the database as well as the software used to connect to it. Before the project started, it was explicitly stated that the implementation of the lexicon server would in some manner provide an extensible approach, in that once the system is designed, it would be able to accept extensions readily. This was realized thanks to the idea of plug-ins, where each component, including the database access classes, hook up into the client application and operate seamlessly without too much complication. This required one common bare bones client capable of accepting both standard user and administrator plug-ins. The former plug-ins together with the common client application are provided for download, while the administrator are kept private. To further provide interaction between users of the system, each standard client application contains an inbuilt rudimentary chat tool.

To augment the rest of the system providing the users with an automated registration facility, and a place where tools and updates can be posted and downloaded, a project website was set up. Apart from providing a registration service, the site also allows the users to unregister, as well as edit their details. The online Maltese dictionary can also be accessed and searched from here, and each entry in the dictionary is linked to similar words thanks to the cluster structures.

A set of tools for manipulating words directly from text files has also been provided. In short, these tools allow the user to cluster words and align individual words either by aligning whole files or by manually supplying the words to be aligned. Distance between words can be found in a similar manner. Statistics about words can be found by examining files for word frequencies and percentages. Although this feature is able to deal with only one file, in the future we will hope to extend this facility to cater for number of text files. A basic word list building tool, able to merge multiple files in one, and applying any rewrite rules supplied by the user, was also provided.

6.1 Further Work

Whereas the aims of the original system have been fulfilled, this project is still in its infancy, and much work can be done to improve and expand it. In this section, we will briefly outline possible upgrades that can be implemented to greatly improve the system, both for the users building the lexicon/dictionary, as well as the 'external' users making use of the system. These updates are categorized and discussed below.

Clustering

Clustering was moved out of the central server onto the client machines, in an intent to relieve the server machine of the precious processing power, which can be used elsewhere. The additional advantages of this approach is that clusters can be created on client machines, and then submitted later to the server, where these are reviewed by appropriate staff, and either accepted or rejected. A downside with this approach is however that a word is not clustered as soon as it enters the database, but has to be considered for clustering by the users building the lexicon.

If the clustering algorithm is further perfected, one might consider the fact of incorporating it also on the server side, since now clustering would be more reliable. There are several possible updates which can be done, but at the time of writing, two reasonable updates are envisaged, and these stem out from the problems faced by the clustering algorithm, discussed in Subsection 3.2.1 on page 39. The problem of *conflicting consonants* could partially be solved if the clustering algorithm was able to access foreign dictionaries like Italian, English and French, so that it would be able to somehow make and informed decision on whether a word is of Romance or Semitic origin. By using suitable weights, the algorithm would possibly be able to determine whether wrong clusters have been formed, in that Semitic and Romance words with overlapping consonants are grouped together. In addition to this, if the dictionaries are of very high quality, and include the stems of particular words, words can be processed prior to the actual clustering algorithm, where the output of this preprocessing phase would be two clusters, one containing Romance words, and the other Semitic words. Then, specialized algorithms, which exploit the characteristics of both languages could be used. For example, the root-based clustering algorithm could be used to cluster words in the Semitic cluster, and simple Levenshtein distance with affix stripping could be used for the second cluster[1].

The second update, which is oriented towards the Semitic part of the language involves using an appropriately constructed electronic dictionary which the algorithm would use to detect irregular/weak verbs.

Other updates that can be suggested include using consonant templates (CV patterns) to try to 'reverse engineer' (Semitic) words, and obtain their base form, which could then be compared to determine the correct cluster. This method, although using already specified (manually built) templates, brings along a number of complications; these are discussed in Rosner et al. (2000). Note that all these updates discussed in this section are in a way not related to our original aim, since the clustering algorithm adopts a knowledge-free approach to clustering. Yet these updates could improve our method drastically.

Framework, Tools and Website

Because of the finite amount of time, certain issues and design decisions regarding the development of the system as a whole, particularly, the framework, were of higher priority than others. Thus, it was made sure that first, the basic requirements of the system were reached, and then, if there was additional time left, other less important issues would be tackled. Unfortunately, no time was left, and therefore, we will suggest additional updates regarding the general framework here.

One small inconvenience that could be caused is the fact that when posted user amendments are either accepted or rejected, there is no way in which they can 'know' this. The original idea was to allow the system to send automatically generated e-mail, explaining whether their contribution was considered or not. In fact, the user name was specifically selected to be an e-mail address, so that should the need for its use arise, no changes to the users table are needed. As already discussed in Section 4.2, the code for generating and sending e-mail is already present, and therefore, this minor change to the framework will greatly improve the user experience.

The text processing tools are very basic for the moment, and these were introduced just to provide the users with the ability of processing raw text files. Additional components that could be provided include implementations of local and global multiple alignment algorithms, as well as extending the statistics tool to support multiple files.

[1]The root-based algorithm could be used on both clusters, since it is able to cluster Romance and Semitic words correctly. However, the initial preprocessing phase would mean that the root-based algorithm will not confuse stems/roots sharing the same consonants.

Additional components that would greatly improve our repertoire of user tools include verb analysis algorithms, able of analyzing the morphology of Semitic verbs (perhaps using grammar rules). Another morphology-related tool would be a fully featured morphology analysis program similar to Goldsmith's *Linguistica*[2]. Since tools like this are only able to analyze Indo-European languages, such a tool must be augmented with the facilities able to also analyze infixes which are present in Semitic languages.

One tangible output of the project, apart from the framework and lexicon itself, is the creation of the first searchable Maltese-English dictionary. The original intent was to try and produce a basic dictionary structure which would however resemble Aquilina's dictionary, in both quality and structure. Of course, we are a long way off, but by the use of our cooperative framework, an evolving dictionary can be built in hopefully lesser time than it would take a single person to complete. Maltese dictionaries are very lacking, especially for the fact that first, the Maltese language is not promoted enough, and secondly, because some of them (for example Aquilina's) cost a fortune. Of course, we are not suggesting that these printed dictionaries should be free of charge. Instead, we are aiming to create a dictionary which apart from being accessible to everyone for free, is more flexible in terms of searching and multimedia support, not only for Maltese people, but also for foreigners, since the promotion of the language also involves the construction and maintenance of a good and reliable dictionary reflecting the current state of the language.

Several projects, such as the EDICT project (Breen, 1999) provides dictionaries which are created in a distributed manner, and made freely available to everyone. Such projects also produce dictionary releases which can be downloaded by software companies, and used to provide more suitable interfaces with additional facilities not present on-line. In our case, an appropriate format must be devised, so that we are able to provide free dictionary releases — in turn these can boost other projects in need of suitable electronic dictionaries[3]. For such purposes, XML can be used to provide a suitable format, since this is readily accessible and supported by most programming languages such as C# and Java. Coupled with this dictionary, an on-line (at least for now) spell checker can also be made available[4].

[2]Linguistica is the implementation of the algorithm discussed by Goldsmith in Subsection 2.2.2 on page 16.

[3]A tool for dumping the contents of the dictionary into either XML or text files has been developed.

[4]An excellent Maltese spell checker project for Linux already exists at:

The website provides basic functionality related to the use of the system, namely, the dictionary, provision of the applications and the user registration process. It would be very useful to include a common language phrase book, such as those found in many sites, aiming to familiarize tourists with the language of the country and common spoken phrases. Of much importance is also the compilation of appropriate help files needed to support the distributed software (refer to Appendix C on page 94 for basic instructions on using the software).

6.2 Conclusion

This project presented a general and extensible framework for the building and maintenance of a computational lexicon for the Maltese language. It also provided the necessary tools for the first distributed construction of both a comprehensive word list, as well as a high quality Maltese dictionary, publicly accessible and freely available. An extensive examination of how structure could be provided to the dictionary was made, and various ideas from articles were used to come up with a suitable clustering algorithm able to cluster Maltese words according to their stem or root.

The framework also provides the necessary APIs that can be used by programmers to invoke the lexicon services. Such a system with suitable provision of programmatic access is a very basic and important building block in natural language systems, such as automatic translation and language understanding. Being the first of its kind, this project had to make use of knowledge-free approaches, since no electronic dictionaries, lexicons or thesauri exists for Maltese. It is hoped that in the future, Maltese-related NLP systems can be developed in a relatively easier manner, since now at least the creation of one fundamental missing component has been tackled. Finally, it is also hoped that by the creation of an evolving dictionary, the Maltese language is kept alive as it has been by others in the past.

`http://linux.org.mt/article/spellcheck`.

Appendix A

Test Set

This appendix includes the cluster test set that was used in Chapter 5, to test both the bi-gram and the root-based clustering algorithms. This set was extracted from the Maltese-English dictionary by Aquilina (1987-1990), and was used as a reference to measure the performance of both clustering algorithms. Since both algorithms rely on a set of affixes that is used in the stemming process, this set of affixes is also provided in this appendix.

A.1 Clusters

xebaħ ($\sqrt{\textbf{XBĦ}}$) xebħ, xebħa, xbieħa, xbiħa, xebbaħ, xebbeħ, xebbieħ, tixbiħ, xiebaħ, mxiebħa, ixxebbaħ, ixxebbeħ, ixxiebeħ, xtiebaħ, mxebbaħ.

sawwat ($\sqrt{\textbf{SWT}}$) msawwat, isawwathom, isawwat, imsawwat, isawwatni, sawwata, sawwatin, sawwati, sawt, swat, sawta, sawtiet, tiswit, issawwat.

rabat ($\sqrt{\textbf{RBT}}$) marbut, torbot, torbotx, orbtu, jorbotni, jorbothom, jorbtux, irbatt, rbattx, rabtuh, rabtet, marbuta, rbata, rbita, rbajjet, rbit, rabta, rabbat, trabbat, rabbata, tarbit, trabbit, trabbita, ntrabat, rtabat, jirtabat.

radam ($\sqrt{\textbf{RDM}}$) radam, mirdum, mardum, mordum, radmu, jordom, irdamtu, nordmok, rdum, rdumi, rdumija, rdumin, rdajjem, radma, ntradam, rtadam.

peċpeċ ($\sqrt{\textbf{PĊPĊ}}$) mpeċpeċ, jpeċpċu, tpeċpeċ, impeċpċin, peċpieċi, peċpieċ, peċpuċa, peċpuċ, peċpieċija, nitpeċpeċ, titpeċpeċ, tpeċpiċ, tpeċpiċa.

opinjoni (opin) opinjonijiet, opinabbli.

oppożizzjoni (oppo) opponent, oppositur, oppositura, oppösituri, oppona, jopponi, topponihx.

għerq ($\sqrt{\textbf{GĦRQ}}$) għeruq, għarraq, mgħarraq, għerejjaq, għerejjeq.

għallaq ($\sqrt{\textbf{GĦLQ}}$) għalaq, igħallqu, mgħallaq, għallaqtni, għallieq, għallieqa, għallieqin, għalliqija, għolliqa, għollieqa, għolliq, tagħliq, tgħliqa, tgħallaq, tgħallieq.

nota (not) noti, noterell, noterella, notabbli, notazzjoni, annotazzjoni, notazzjonijiet, annotazzjonijiet, notament, notamenti, notevoli, notifikazzjoni, notifikazzjonijiet, innota, innotat, tinnota, jinnota, notifika, notifiki, innotifika, innotifikat, notifikat, innotifikar, notifikar.

nom (nom) nomi, nominali, nominalistiku, nominalistika, nominalistiċi, nominalizmu, nominaliżmi, nominalment, nominattiv, nominattiva, nominattivi, nominazzjoni, nominazzjonijiet, nomna, nomina, nomni, nomini, innominabbli, innomina, innomna, innominat, innominar, innomnat, innomnar.

norma (norm) normi, normal, normali, anormali, normalment, normalità, anormalità, normalizzazzjoni, normalizzazzjonijiet, innormalizza, innormalizzat, normalizzat, normalizzar, innormalizzar.

niżel ($\sqrt{\textbf{NŻL}}$) inżel, nieżel, minżul, nżilt, jinżel, niżelhom, niżlux, tinżillix, jinżillix, jinżlu, nieżel, nieżla, niżla, nżieli, nżuli, niżli, nżulija, niżlija, niżlin, niżżel, mniżżel, niżżluh, inniżżlu, iniżżel, iniżżillek, jniżżel, niżżiel, niżżiela, niżżieli, niżżielin, niżżelija, tinżil, tinżila, tniżżel, tniżżil, tniżżila, minżel, mnejżla, minżla, nżul.

nibet ($\sqrt{\textbf{NBT}}$) minbut, jinbet, nibtulha, inbitt, niebet, niebta, nibtin, nbit, nibta, nibtiet, nbieta, nbiet, nbieti, nbietija, nibbet, nebbet, mnibbet, mnebbet, i-nibbet, inebbet, nebbiet, nibbiet, nibbieti, nebbieta, nibbieta, nebbieti, tinbit, tinbita, tnebbet, tnibbit, tnibbita.

modern (modern) mudern, moderna, moderni, modernament, modernista, modernisti, modernità, moderniżmu, modernizzatur, modernizzatura, modernizzaturi, modernizzazzjoni, modernizzazzjonijiet, immodernizza, immoderna, modernizzat, modernizzar, immodernizzat.

miżerja (miżer) miżerji, miżeru, miżera, miżeri, miżerabbli, miżarabilment, miżerjament.

nies (—) stienes.

miraklu (mirak) mirakli, mirakoluż, mirakuluż, mirakolożament, mirakulat.

mesaħ ($\sqrt{\text{MSĦ}}$) mimsuħ, imsaħ, nimsaħ, timsaħ, mesħet, mesħitu, msiħ, mesħa, mesħiet, msieħ, masħa, masħiet, messaħ, messieħ, timsiħ, tmessaħ, timsiħa, ntmessaħ.

mertu (mert) meritu, merti, meriti, meritevoli, meritevolment, meritokrazija, meritokratiku, immerita, demertu, demerti.

meraq ($\sqrt{\text{MRQ}}$) merqtu, merq, merqiet, merraq, mmerraq, mmerrqa, merrieqi, merriqija, merriqin, tmerraq, tmerriq, tmerriqa.

maxat ($\sqrt{\text{MXT}}$) mimxut, maxatlu, moxt, moxtijiet, mxat, maxxita, mxit, mxata, maxxat, immaxxat, timxit, timxita, tmaxxat, tmaxxit, tmaxxita, ntmaxat, mtaxat, maxta.

marad ($\sqrt{\text{MRD}}$) nimrad, mard, marid, marida, morda, marrad, mmarrad, tmarrad, imarradni, marrada, marradin, marradi, mrajjed, marradija, tmarrid, mardet.

żgħir ($\sqrt{\text{ŻGĦR}}$) żgħira, żgħar.

żifen ($\sqrt{\text{ŻFN}}$) miżfun, żifna, jiżfen, niżfen, żfin, żifniet, niżfnu, żfejna, żeffen, iżeffinhom, iżeffen, iżeffinni, imżeffen, żeffien, żeffiena, żeffieni, żeffienija, żeffienin, tiżfin, tiżfina, iżżeffen, jiżżeffnu.

xewwex ($\sqrt{\text{XWX}}$) mxewwex, ixewwex, xewwiex, xewwiexa, xewwiexin, xewwiexi, xewwiexija, tixwix, tixwixa, ixxewwex, txewwex.

xabbat ($\sqrt{\text{XBT}}$) mxabbat, xabbata, xabbatin, xabbati, xabbatija, xabbatur, xabbatura, xabbaturi, tixbit, tixbita, ixxabbat, ixxabtu.

wiret ($\sqrt{\text{WRT}}$) jiret, mirut, wiritha, wirt, wirta, werret, mwerret, werriet, werrieta, werrietin, werrieti, werrietija, jitwerrtu, twerrit, twerrita, ntiret.

wikka ($\sqrt{\text{WKJ}}$) wekka, mwikki, twikki, wikkielek, wekkej, wekkejja, wekkejjin, tiwkkijja, twekka, twekkija.

wieħed ($\sqrt{\text{WĦD}}$) waħda, waħdiet, uħud, wħud, waħdi, waħdu, waħdek, waħedha, waħidhom, waħdien, waħdieni, waħdenija, waħdenin, waħħad, mwaħħad, teħwid, tewħidha.

werżaq ($\sqrt{\text{WŻRQ}}$) mwerżaq, twerżaq, twerżiq, iwerżaq, werżieq, wirżieq, urieżaq, wrieżaq, werżieqi, werżieqija, werżieqin, wrejżaq, werżiqa.

alterazzjoni (alter) alterazzjonijiet, alterabbli, inalterabbli, alterat, altera, alterar, taltera.

antik (antik) antika, antiki, antikalja, antikament, antikità, antikwarju, antikwarji, antikwarjat, antikwata, antikwati, antikwalja.

awtorità (awtor) awtoritajiet, awtoritevoli, awtorevolment, awtoritarju, awtoritarja, awtoritarji, awtorizzazzjoni, awtorizzazzjonijiet, awtorizza.

bakar ($\sqrt{\text{BKR}}$) mibkur, jobkor, bkur, bkura, bkuri, bkurija, bkurin, bkir, bkieri, bikri, bikrija, bikrin, bokra, bakkar, mbakkar, bakkari, bakkara, bakkarin, bakkarija, tibkir, tibkira, tbakkir.

bambal ($\sqrt{\text{BMBL}}$) bambalulu, bambalielhom, bambalulhek, tbambila, ibambal, tbambil.

baram ($\sqrt{\text{BRM}}$) mibrum, mibruma, jobrom, jobormok, jobromlu, brim, barma, barram, ibarram, barrama, barramin, tbarram, tbarrim, tbarrima, nbaram, btaram, mbaram.

xedaq ($\sqrt{\text{XDQ}}$) ixdqa, xdieq, xedqajn, xedqejn, xdiq, xdiqi, xedqu, xdejjaq.

webbel ($\sqrt{\text{WBL}}$) mwebbel, webbiltni, webbilni, webbiel, webbiela, webbeielin, twibbel, jitwebblu, twebbilt, twebbil.

warrab ($\sqrt{\text{WRB}}$) mwarrab, imwarrab, iwarrabna, warraba, warrabija, warrabin, twarrib, twarriba, twarrab.

waqaf ($\sqrt{\text{WQF}}$) jieqaf, ieqaf, tieqaf, waqqfet, iqaflu, waqaflu, wieqaf, wieqfa, wiqfin, waqfa, waqfien, wqif, waqqaf, waqqafni, jwaqqaf, waqqaftu, twaqqaf, twaqqif.

textex ($\sqrt{\textbf{TXTX}}$) mtextex, mtextxa, textiexi, textiexija, textiexin, textix, textixa, textuxa, textuxiet.

taqab ($\sqrt{\textbf{TQB}}$) mitqub, tqib, toqba, toqbiet, tqajba, tqajbiet, toqbi, taqqab, m-taqqab, taqqabtli, taqqaba, titqib, ittaqqab, ntaqab.

varjetà (varj) varjetajiet, varju, varja, varjabbli, invarjabbli, varjabilità, varjanti, varjazzjoni, varjazzjonijiet, varjar, varjat, ivarja, tvarja.

seħer ($\sqrt{\textbf{SĦR}}$) sħarijiet, isħra, saħħar, msaħħar, ssaħħritu, saħħara, sħaħar, tisħir, tisħira, issaħħar, nseħer.

qarad ($\sqrt{\textbf{QRD}}$) maqrud, oqrodha, qrid, qard, qarda, qardiet, qarrad, mqarrad, qarried, qarrieda, qarriedin, qarredija, qordiena, qardiena, qurdiena, taqrid, tqrida, nqarrad, nqorod.

qatel ($\sqrt{\textbf{QTL}}$) maqtul, toqtol, toqtolni, noqtolok, joqtol, toqtlu, qatlu, joqtolhom, joqtolni, joqtlok, qatlitu, toqtlok, oqtolni, oqtlu, qatilha, qtil, qatla, qatlet, qattel, mqattel, qattiel, qattiela, qattielin, qattieli, qattielija, taqtil, taqtila, tqattel, taqttil, tqattila, tqatel, tqatil, nqatel, jinqatlu.

lewn ($\sqrt{\textbf{LWN}}$) lwien, lewnijiet, lewwen, mlewwen, tilwin, tilwina, tlewwin, tlewwen.

lagħab ($\sqrt{\textbf{LGħB}}$) milgħub, jilgħabu, jilgħab, nilgħab, jilgħabhom, tilgħab, tilgħabli, lagħbuh, lagħablu, lagħabhomlu, jilgħabha, lagħabtlu, tilagħabha, lagħba, logħbiet, lgħajba, lgħajbiet, logħob, logħobok, lagħbi, lagħbija, milgħaba, milgħabiet, tliegħeb, ntlagħab, milgħab.

laħlaħ ($\sqrt{\textbf{LĦLĦ}}$) mlaħlaħ, ilaħlaħ, imlaħlaħ, imlaħalħa, laħlieħ, laħlieħa, laħlieħin, laħlieħi, tlaħlaħ, nitlaħalħu, jitlaħlaħ, laħliħ, tlaħliħ, laħliħa, tlaħliħa.

konċett (konċe) konċetti, konċettiv, konċetta, konċettwali, konċettwaliżmu, konċettwalista, konċettwalisti, konċepibbli, inkonċepibbli, ikkonċepxxa.

keskes ($\sqrt{\textbf{KSKS}}$) mkeskes, ikeskes, imkeskes, kiskes, keskiesa, keskisiet, keskies, keskiesin, keskiesi, tkeskes, jitkesksu, tkeskis, tkeskisa.

insinwa (insinw) insinwi, insinwat, insinwar, jinsinwa, insinwazzjoni, insinwazzjonijiet.

ħolom ($\sqrt{\textbf{ĦLM}}$) moħlum, nħolom, ħlomt, joħlom, toħlom, ħlomtha, joħlomha, ħolm, ħolma, ħallem, mħallem, ħallemni, ħellemha, tħallmu, taħlim, taħlima, ħelliem, ħelliema, ħalliemin, ħelliemi, ħalliemija, ħelliemin, tħallem, tħallim, tħallima, noħlom.

ħadem ($\sqrt{\textbf{ĦDM}}$) maħdum, nħadem, jaħdem, tħadem, taħdima, tħadmu, jaħdmek, jaħdmini, taħdmu, ħadimlek, ħidma, ħdim, ħadma, ħaddem, ħaddimtu, inħaddmek, ħaddmu, iħaddmu, ħaddimhom, ħaddimtha, ħaddiem, ħaddiema, ħaddiemi, ħaddiemin, taħdim, tħaddem, tħaddim, tħaddima, naħdem.

hemeż ($\sqrt{\textbf{HMŻ}}$) mhemuż, mhemużin, ahmeż, hmiż, hemża, hemmeż, hemmież, hemmeża, hemmeżin, hemmeżija, hemmeieżi, themmeż, themiż, mahmież.

gżira (gżr) gżiriet, gżejjer, gżejra, gżejri, gżejriet, gżejrija, gżejrin.

gerfex ($\sqrt{\textbf{GRFX}}$) mgerfex, gerfexhomli, imgerfex, nitgerfex, tgerfex, jgerfxu, gerfiex, gerfiexa, gerfiexin, gerfiexi, gerfux, gerfuxa, gerfuxiet, grifiex, gerfix, gerfixa, tgerfix, tgerfixa.

ġdid (—) ġdida, ġedded, mġeded, ġeddulu, ġeddu, ġeddied, ġeddieda, ġeddiedin, ġeddiedi, tiġdid, tiġdida, iġġedded.

ġabar ($\sqrt{\textbf{ĠBR}}$) miġbur, iġbor, ġabru, tiġborhiex, tiġbrux, miġburin, jiġborhuli, jiġbrilha, niġbor, ġabruh, ġbarthom, ġabarhom, ġbir, ġabra, ġabriet, ġbira, ġabbar, mġabbar, ġabbara, ġabbarin, ġabbarija, ġabbari, tiġbir, tiġbira, iġġabbar, nġabar, tinġabar, inġabret, ġabrithu, jinġabar, maġbar, miġbra.

fetaq ($\sqrt{\textbf{FTQ}}$) miftuq, fetqet, joftoq, fetqitu, ftuq, fetqa, ftejqa, fettaq, nfettqu, fettieq, fettieqa, fettieqin, fettieqi, fettuqa, ftietaq, tiftiq, tiftiqa, tfettiq, nfetaq.

furketta (furkett) frieket, furkettata, furkettati, furkettun, furkettuni, iffurkettja, iffurkettjat, iffurkettar, ffurkettat.

felaħ ($\sqrt{\textbf{FLĦ}}$) mifluħ, jiflaħ, niflaħ, jiflaħx, tiflaħ, flaħna, niflaħx, jifilħux, felħ, felħi, felħa, felħin, fellieħ, fellieħa, fellieħin, felħan, felħana, felħanin, nfelaħ.

fadal ($\sqrt{\textbf{FDL}}$) fadallek, fadalx, fdal, fdalijiet, fadla, fdala, faddal, mfaddal, tfaddal, faddala, faddalin, faddalija, tifdil, tifdila, tfaddil.

fehem ($\sqrt{\textbf{FHM}}$) mifhum, tifhem, fhimtni, jifhimx, nifhem, jifhem, jifhmu, fehim, fehma, fehmiet, fiehem, mfiehem, fehiem, fehiema, fehiemi, tfiehem, tfehim, tfehima, nfehem, nftehem, ftiehem, nftiehem, ftehim, ftehima.

eżerżizzju (eżerċ) eżerċizzji, eżerċenti, eżerċita, eżerċitat, eżerċitar, jeżerċita.

evolut (evol) evoluta, evoluti, evolva, evoluttiv, evoluzzjoni, evoluzzjonijiet, evoluzzjonista, evoluzzjonisti, evoluzzjoniżmu, evolventi, evolvi.

dendel ($\sqrt{\textbf{DNDL}}$) mdendel, imdendel, dendluh, dendil, dendila, dendiel, dendiela, dendielin, dendul, dendula, denduliet, denduli, iddendel, ddendelna, iddendlet.

deċiżjoni (deċiż) deċiżjonijiet, indeċiżjoni, deċiż, deċiża, deċiżi, indeċiżi, indeċiża, deċiżiv, deċiżiva, deċiżivi, deċiżur, deċiżuri, deċiżament, iddeċieda.

daqq ($\sqrt{\textbf{DQQ}}$) idoqq, jdoqq, ndoqqu, doqqli, ddoqqx, jdoqqlix, iddoqqlu, ddoqqhomlu, doqqhomli, daqqu, daqqlu, daqqiet, daqqa, dqajqa, daqqaq, iddoqq, daqqaqa, daqquqa, daqquqiet, daqqiq, iddaqqaq, jiddaqqaq, iddaqqaqt, tiddaqqaq, ndaqq, indoqq, mdoqq.

daħal ($\sqrt{\textbf{DĦL}}$) dieħel, daħlet, dħalna, daħlu, jidħlux, jidħollokx, tidħollix, jidħlu, dħalt, daħlilna, daħlilha, tidħol, jidħollok, tidħolli, jidħol, daħlitlu, deħlin, dħul, dħuli, dħulija, dħulin, dħula, daħla, daħliet, daħħal, mdaħħal, daħħalt, idaħħal, daħħalha, jdaħħal, jdaħħalx, idaħħalha, ddaħħalnix, daħħaltni, idaħħalhom, daħħalin, daħħali, tidħil, tidħila, iddaħħal, ndaħal, tindaħallix, tindaħal, ndħil, midħal, midħla.

boqqa ($\sqrt{\textbf{BQQ}}$) boqoq, boqqiet, bqajqa, bqajqiet, baqqa, mbaqqi, tbaqqix, tbaqqiqa, tbaqqija, tbaqiq, tbaqija.

bierek ($\sqrt{\textbf{BRK}}$) mbierek, jbierek, imbierek, tbierek, tberik, tberika, barka, barkiet.

silet ($\sqrt{\textbf{SLT}}$) mislut, siltu, mislutin, nisiltu, jisiltek, misluta, misultin, silta, siltiet, slejta, slejtiet, sellet, msellet, selliet, sellieta, sellietin, sellieti, sellietija, tislit, tislita, issellet, tissellet, issielet, nsilet, jinsilet, stilet.

biegħ ($\sqrt{\textbf{BJGĦ}}$) mibjugħ, ibigħ, tbigħhieli, begħet, jbigħek, biegħet, jbegħiha, bejgħ, biegħa, bejjiegħ, nbiegħ.

batal ($\sqrt{\textbf{BTL}}$) mibtul, btalet, ibtal, battal, mbattal, imbattal, battala, battalin, tabtil, tabtila, btala, btajla, btajliet, tbattal, jitbattlu, tbattlu, tbattil, tbattila.

baħar ($\sqrt{\textbf{BĦR}}$) ibħra, bħar, bħur, bħajjar, bħajra, bħajjer, baħri, baħrija, baħrin, tibħir, tibħira, baħħar, baħħara, tbaħħar, tbaħħir.

ċaflas ($\sqrt{\textbf{ĊFLS}}$) mċaflas, ċaflastli, ċaflastha, ċaflis, ċafles, ċaflasa, ċaflasin, ċaflasi, iċċaflas, ċaflusi, ċaflusija ċaflusin.

ċekċek ($\sqrt{\textbf{ĊKĊK}}$) mċekċek, jċekċek, ċekċku, ċekċikhielu, ċekċiklu, iċekċek, ċekċkuha, ċekċik, ċekċika, ċekċikiet, ċekċiek, ċekċieka, ċekċiekin, ċekċieki, iċċekċek, ċekċikija.

ċekken ($\sqrt{\textbf{ĊKN}}$) mċekken, ċekkinha, ċekkien, ċekkiena, ċekkieni, tiċkin, ċekkin, tiċkina, iċċekken, tiċċekken, ieċċekken, ċkien, jiċkien, ċokon, ċkunija, ċkejken, iċken, ċkejkun, ċkejkuni, ċkejkuna, ċkejkunin.

A.2 Affixes Used for Stemming

Prefixes
d-, id-, il-, im-, in-, ix-, iċ-, iġ-, iż-, ir-, is-, j-, m-, n-, nt-, s-, st-, t-, z-, ċ-, ġ-, ż-.
Suffixes
-a, -abbli, -ajn, -ajt, -ajtu, -ajżer, -an, -ant, -anza, -ar, -at, -ata, -ati, -attiv, -atur, -aw, -ejn, -ejna, -ejt, -ek, -ell, -ella, -elli, -erija, -et, -ew, -h, -ha, -hom, -i, -ien, -iet, -ija, -ijiet, -iku, -ilità, -iltı, -in, -ist, -ista, -istika, -istiku, -istiċi, -it, -ita, -ittivament, -ittività, -iv, -ivament, -ixxa, -izza, -iżmu, -ja, -jiet, -ju, -k, -ki, -kom, -liżma, -liżmi, -liżmu, -lment, -ment, -menti, -na, -ni, -ok, -oloġija, -ometru, -t, -tejn, -ttiv, -ttiva, -ttivi, -tu, -tur, -tura, -turi, -u, -x, -zzjoni, -zzjonijiet, -ċi.

The table shown above displays the list of prefixes and suffixes that was used during the tests carried out in Chapter 5. It must be noted that this set of affixes is not exhaustive, and it contains enough affixes so that the bi-gram and the root-based clustering algorithms could be tested. Therefore, although this affix set is appropriate for the the dictionary test, for the second set of tests (the bible text test), some of these were applicable, while others were not. Also, as already discussed in Definition 3.1 on page 35, we do not employ a recursive morphological structure.

Appendix B

Maltese in a Nutshell

This appendix attempts to provide a very brief introduction the the Maltese grammar. It does not try to give a full coverage of the grammar, but rather highlight the main points in how words and verbs are formed, allowing one to see to what extent the Maltese language is 'complex' in relation to other languages.

Maltese is the official language of the Maltese islands, and is spoken by a third of a million people inhabiting the island, as well as by immigrants scattered in Austraila, Canada and the USA. Maltese is classified as a dialect of Arabic, which however includes various differences not found in other dialects deriving from Arabic. In addition, Maltese also incorporates a long list of Romance and Anglo-saxon words. Maltese thus combines the characteristics of Southern Europe and North Africa, reflecting its geographic location and it is precisely for this reason, that Maltese stands out, and demands special attention.

B.1 Basics

B.1.1 The Maltese Alphabet

Although Maltese is a Semitic language, it uses Latin and additional modified Latin characters for its alphabet. It was codified in the 1920s by the 'Akkademja tal- Malti' (formerly known as the 'Għaqda Kittieba Maltin'). The alphabet is made up of 30 letters, two of which, the 'ie' and 'għ' are made up of two characters each. The vowels are 'a', 'i', 'u', 'e', 'o' and 'ie', the first five of which can be either pronounced as short or long vowels; the vowel 'ie' is always pronounced as a long vowel. The Maltese alphabet is as follows:

A a, B b, Ċ ċ, D d, E e, F f, Ġ ġ, G g, GĦ għ, H h, Ħ ħ, I i, IE ie, J j, K k, L l, M m, N n, O o, P p, Q q, R r, S s, T t, U u, V v, W w, X x, Ż ż, Z z.

B.1.2 Roots

The characteristic of Semitic languages (unlike Romance) is that vowels change easily or in some cases, disappear completely. On the other hand, the consonants do not change so often (not even their relative position), and this is especially true for the radical consonants (i.e. consonants of the root). The root consonants are usually those which appear in the simplest word form, which is usually the *mamma* (the first verb form, past, 3rd person, masculine) or the simplest noun (in the case where a verb does not have a first verb form). Thus for example, the roots of *kiteb* (wrote), *qatel* (killed), *mar* (went) and *baħar* (sea) are k-t-b, q-t-l, m-w-r and b-ħ-r respectively. Note that the the verb *mar* is a weak verb, that is, even though the consonant 'w' is not found in this form, is can be found in variants of this word, such as *mawra* (a cruise/journey). The consonants 'j' and 'w' very often appear and disappear in different forms of the same word, and are called weak consonants. In the case of the word *baħar*, which is a noun, no first verb form exists, and therefore, we use the noun to extract root consonants. It is common practice that for weak verbs, one considers different forms of that verb so as to try to determine which consonants are missing.

The majority of the Maltese verbs are triliteral, meaning that their root contains three radical consonants. However, there exists also a quite large number of quadriliteral verbs, together with a very small amount of verbs which have a root made up of only two consonants. This last set of verbs must not be confused with triliteral verbs having a weak consonant, since in their origin, these have exactly two radical consonants, e.g. bin (son).

B.1.3 The Article

In its simplest form, the article is the letter 'l-' which is placed in front of a noun to make it a determiner. It is possible for the article to take an additional 'i' (called the *vokali tal- leħen*) in front of the 'l-' so that it is pronounced correctly, such as in

il-ġurija (the jury)

However, the addition of an extra 'i' is not performed if the word preceding the article ends in a vowel:

kellu l- flus (he had the money)

In the case where the article appears before the consonants 'ċ', 'd', 'n', 'r', 's', 't', 'x', 'ż', 'z', the 'l' in the article changes into one of the respective consonants. If appropriate, the article is also preceded by an 'i'.

iċ- ċekċieka (the rattle), *id- dgħajsa* (the boat), *in- nanu* (the dwarf), *ir-re* (the king), is- serp (the snake), *it- taxxa* (the tax), *ix- xita* (the rain), *iż- żahar* (the blossoms).

B.2 Nouns

B.2.1 Pronouns

The personal pronouns are:

Singular	**Plural**
jiena/jien (I/me)	aħna (we/us)
inti/int (you)	intom (you)
huwa/hu (he)	huma (they)
hija/hi (she)	huma (they)

When we want to produce the negative of these, we precede them by the word 'ma' and add the suffix 'x'. However, in this case the 'a' of 'ma' is written as an apostrophe (') since two vowels cannot lie near each other[1]. Thus the above are written as:

Singular	**Plural**
m'iniex (I'm not)	m'aħniex (we're not)
m'intix/m' intx (you're not)	m'intomx (you're not)
m'huwiex/m'hux/mhux (he's not)	m'humiex (they're not)
m'hijiex/m'hix/mhix (she's not)	m'humiex (they're not)

[1]Note that the letter 'h' is followed by a vowel; however we disregard this letter and consider the letter following it, which in these particular cases is a vowel. The 'h' is not pronounced in the beginning and middle of words, however in some cases it is pronounced as 'ħ' at the end of the word.

Unlike English and other languages, in Maltese, pronouns are attached directly to the verb or noun, and thus a given word may refer to different persons (by attaching a number of suffixes) at the same time. The negative is also attached after all pronominal suffixes. The pronominal suffixes are the following:

Pronoun	**Suffix**
jien	i, ja, ni*
inti	k, ek, ok
huwa	u, h, hu*
hija	ha, hie, hi
aħna	na, nie
intom	kom
huma	hom

The suffix ‘hu’ is never used at the end of the word; in fact in Maltese *no* word ends in ‘hu’. Also, the use of the suffix ‘ni’ is only permissible at the end of verbs. Using the table above, complex expressions can be formed, for example:

my thing = *tiegħ-i*(1st. pers.)
he did not find his thing = *ma*(neg.) *sab*(found)-*hi*(3rd. pers.)-*lu*(3rd. pers.)-*x*(neg.)

B.2.2 Nouns

While in Maltese, adjectives have both plural and singular, nouns have singular, plural, collective and the so called *għadd imtenni* (doubling).

The collective, as its name implies, is a singular word which however shows a collection of objects, like *dubbien* (flies) and *larinġ* (oranges). If we add and ‘a’ at the end of these kinds of words, we are able to obtain the feminine, e.g. *larinġa* (orange). In addition, in Maltese, we have the plural of the plural, in which case the collective *dubbien* becomes *dubbiniet*[2].

The *għadd imtenni* is used to denote things that occur in pairs, like *bajdiet* (two eggs), *sentejn* (two years), *dudejn* (two worms). If the plural is not ‘imtenni’ then, it is either sound or broken. The sound plural is called so because it just adds a suffix to the word, without breaking its structure; conversely the broken plural does not add

[2]Note that any word can only contain a maximum of one ‘ie’ in it.

any suffixes, but changes the structure of the word (there are no rules for construction of the broken plural). In some cases, words can have both a sound and broken plural:

tapit (carpet), *tapiti* (carpets, sound), *twapet* (carpets, broken)
furketta (fork), *furketti* (forks, sound), *frieket* (forks, broken)

Both of the above are Romance words, and they can take the broken plural which is usually associated with Semitic languages, as Romance languages do not have the notion of this type of plural. We can see how such loan words are immediately integrated seamlessly into the Maltese morphology. In the case of Semitic words, they can either take the sound or broken plural, but not both:

qarib (relative), *qraba* (relatives)
baħar (sea), *ibħra* (seas)

The sound plural is built using the following rules:

1. Addition of 'ijiet' after a word: *truf* (edge); *trufijiet* (edges), *bajja* (bay); *bajjiet* (bays).

2. Addition of 'ien' or 'an' after a word: *nar* (fire); *nirien* (fires), *qiegħ* (bottom); *qiegħan* (bottoms).

3. Addition of 'a' after a word, which can either show the plural or the 3rd person feminine: *giddieb* (liar); *giddieba* (female liar/liars), *sewwieq* (driver); *sewwieqa* (female driver/drivers).

4. Addition of 'in' after a word: *qaddis* (saint); *qaddisin* (saints).

5. Addition of 't' at the end of a word, providing the word is feminine: *kewkba* (star); *kewkbiet* (stars).

B.2.3 Diminutives

The diminutive indicates smallness in the meaning of words. Some of the diminutives are formed by adding an 'a' after a word, while others add a 'j' as an infix, for example:

ġnien (garden); *ġnejna* (small gardern), *id* (hand); *wejda* (small hand),
ħobża (loaf); *ħbejża* (small loaf), *triq* (road); *trejqa* (small road/path),
xiħ (old man); *xwejjaħ* (small dear old man).

B.3 Verbs

B.3.1 Types of Verbs

Maltese verbs are categorized as sound or weak. Sound verbs have three consonants in their mamma which are neither 'j' nor 'w'. If the last two consonants of the mamma are the same, then the verb is known as *trux* (deaf). Weak verbs have one of the weak consonants in their mamma, and are categorized into three groups:

- *xebbihin*, which have the first radical consonant 'w' (there is one special case involving 'j', *jassar*): *wiret* (inherited).
- *moħfijin*, in which their second radical consonant appears either as a 'w' or a 'j': *miet* (died), *tar* (flew), *sar* (cooked).
- *neqsin*, which have their third radical consonant 'j': *mela* (filled), *ġera* (ran), *xela* (accused).

B.3.2 The Verb Forms

The verb forms are very important, and are used to change the base form of a given verb. There are ten verb forms in Maltese, but not all verbs appear in all ten forms. The first form is the main verb (mamma). Note that for some verbs, the first form does not exist; this is common in verbs deriving from a noun. Nowadays, the fourth form is no longer used, thus we are left with the eight forms shown below:

The second form Verbs duplicate their second radical consonant: *nefaħ* $\longrightarrow$ *neffaħ.*

The third form Verbs lengthen the first vowel occurring in the first verb form *qagħad* $\longrightarrow$ *qiegħed.*

The fifth form Adds a 't' infront verbs of the second form: *neffaħ* $\longrightarrow$ *tneffaħ.* If however, the verb starts with 'ċ', 'd', 'ġ', 's', 'x', 'ż' or 'z', then the 't' changes into the respective consonant, thus, we do not say *tċarrat* but *ċċarrat.*

The sixth form Adds a 't' infront verbs of the third form: *qiegħed* $\longrightarrow$ *tqiegħed.*

The seventh form Verbs add either a 'n' or 'nt' in front of verbs of the first form, or 'n' infront of, and 't' after the first radical consonant of verbs in the first from. Thus: *qatel* $\longrightarrow$ *nqatel*, *lewa* $\longrightarrow$ *ntlewa*, *silet* $\longrightarrow$ *nstilet.*

The eight form These verbs add a 't' after the first radical consonant of verbs in the first form: *nesa* $\longrightarrow$ *ntesa.*

The ninth form Verbs in the first form drop the first occurring vowel in the word, and then, lengthen the remaining ones: *sebaħ* $\longrightarrow$ *sbieħ*, **mejel* $\longrightarrow$ *miel.*

The tenth form Adds an 'st' infront of verbs: *stienes*, *stejqer.*

In the case of quadriliteral verbs, they can be either main or derived. They are derived by adding a 't' in front of the main verb, *laħlaħ* $\longrightarrow$ *tlaħlaħ*, Again, for *dendel* we do not say *tdendel* but *ddendel.*

B.3.3 Verb Conjugates

When we conjugate verbs in the present tense, we prefix them as shown below:

Singular		**Plural**	
jiena (I/me)	Noqtol (I kill)	aħna (we/us)	Noqtlu (we kill)
inti (you)	Toqtol (you kill)	intom (you)	Toqtlu (you kill)
huwa (he)	joqtol (he kills)	huma (they)	Joqtlu (they kill)
hija (she)	Toqtol (she kills)	huma (they)	Joqtlu (they kill)

The imperative in the present tense is as follows: inti *oqtol* (you kill, sing.), intom *oqtlu* (you kill, pl.)

Conjugates in the past tense attach the following suffixes as shown:

Singular		**Plural**	
jiena (I/me)	qtilT (I killed)	aħna (we/us)	qtilNA (we killed)
inti (you)	qtilT (you killed)	intom (you)	qtilTU (you killed)
huwa (he)	qatel (he killed)	huma (they)	qatlU (they killed)
hija (she)	qatlET (she killed)	huma (they)	qatlU (they killed)

Note: This appendix was partially written with continuous reference to the books by the Academy of Maltese (1998) and Cachia (1994), which are two excellent resources on the Maltese grammar. It is recommended that these books are consulted should the need for further details arise.

Appendix C

Using the Software

This appendix provides instructions on how the system should be installed on a new host, and also gives an outline of the functions which are available to the user. The software can be found on the CD accompanying this report, and includes also the versions of MySql and .NET connector driver that were used during the development of the system. All directories that will be mentioned are with reference to the CD.

C.1 Setting up the Lexicon Services on a new Host

Before any of the applications can be used, the .NET framework together with the Internet Information Services (IIS) server must be installed. The .NET framework is required by all applications, since any code that has been written, specifically targets the .NET framework instead of the operating system directly. The IIS web server is needed so that the system is able to host not only the website but also any web services that are to be deployed. The .NET framework is freely available, and can be downloaded from Microsoft's website. IIS is available on the WindowsXP distribution, and should be set up before the .NET framework is installed. Instructions on how these can be installed are found on the vendor's website and WindowsXP help. Once both have been set up, the following instructions must be followed in order:

Installing MySql and the .NET Connector

Both the MySql database server as well as the .NET connector can be downloaded for free from the MySql website (`http://www.mysql.com`). Ideally, MySql is installed first, followed by the installation of the .NET connector. Step-by-step instructions are provided by the installers, and therefore, no additional instructions are given here. It is recommended that the default directory settings are used.

Creating the Lexicon Database

Once the database server is set up, the lexicon database, together with all its tables must be installed before the system can be used. For convenience, an SQL script file is provided (`binaries\lexicon.sql`), and this should be called from the MySql command prompt, as follows:

```
prompt> mysql -u root -p < lexcion.sql
```

When MySql asks for the password, the password which was specified during the installation of MySql (i.e. the root password) must be specified. Once the SQL script file is processed, all the tables and necessary constrains, together with a user identified as ‘client’, are set up. This user is used by the application to access the database, and is given enough privileges to read and write from the database tables.

Installing the Administrator Software

All the applications have been conveniently packaged in setup files to allow easy deployment of the software on any host machine. For the system to be operable, only the website and chat server are required. However, it is recommended that the web service and a bare bones client set up with the administrator plug-ins are also installed.

The website installation files can be found at `binaries\websiteSetup`, and it is recommended that that default options are used. The web service can be installed from `binaries\dictQueryWebserviceSetup`, while the standard (without plug-ins) eLexi application can be installed from `binaries\eLexiSetup`. Once installation is over, administrator plug-ins which are located in `binaries\adminPlugins` can be copied to the `..\eLexi\plugins` directory. The chat server can be installed from `binaries\chatServerSetup`.

Installing the Standard User Software

The software for the standard user has been appropriately packaged so that both the basic eLexi application together with the standard user plug-ins are installed automatically. The setup files for this distribution can be found in `binaries\eLexiUserSetup`.

In addition to the main eLexi client application, the Internet Explorer dictionary bar provided in `binaries\dictionaryBarSetup` may be installed to provide Internet Explorer with the dictionary searching facility. Once the bar is installed, it *must* be activated so that it is accessible from the *View* $\longrightarrow$ *Toolbars* menu of Internet Explorer.

Note that should *any* application require removal, it can be safely un-installed from the Windows control panel.

C.2 Software Overview

This section describes briefly the plug-ins included, and provides basic instructions on their use. The compilation of an extensive online help system was impossible to carry out due to the lack of time and also because such a job requires a significant amount time to complete. Nevertheless, we have tried to capture the main aspects of the system and provide a concise reference manual.

C.2.1 The Administrator Plug-in

Before the application can be used, the user must possess a valid login which can be obtained from the project website. This login is used to log into the application and get access to the tools installed. Once login is completed successfully, the administrator tools can be accessed from the *Tools* menu, as depicted in Fig. C.1.

All four dictionary management tools work on the same principle, and contain functions to process amendments posted by users and access the database directly. Since all of these enclose virtually the same functionality, we will tackle the most complex component of them all: the dictionary management tool.

The dictionary management tool show in Fig. C.2 allows the user to view and *directly* modify entries in the dictionary. Words in the dictionary can be introduced as long as they already exist in the basic word list. This is done because the word list is the basic collection of words, and any word that enters or leaves the lexicon must first be added to this basic word list. This implies that all other tables such as the clusters and the dictionary tables are controlled by the word list and therefore, should one want to remove (or update) a word from the list, all occurrences of this word (whether in the dictionary or in some cluster) are removed (or updated) automatically.

Words can be viewed on a prefix basis, which means that records are retrieved if

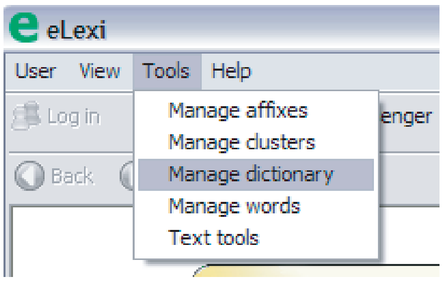
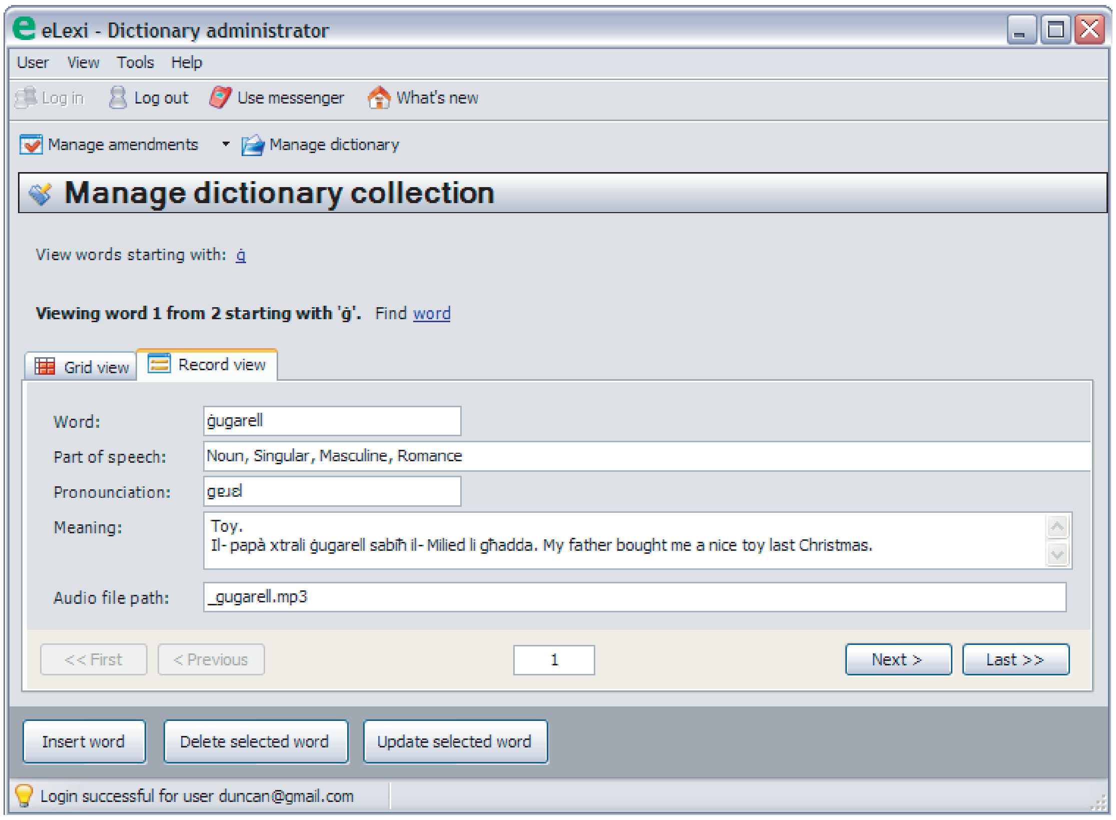


Figure C.1: The administrator's tools menu.



Manage dictionary collection

View words starting with: ġ

Viewing word 1 from 2 starting with 'ġ'. Find word

Word:	ġugarell
Part of speech:	Noun, Singular, Masculine, Romance
Pronounciation:	geɹel
Meaning:	Toy. Il- papà xtrali ġugarell sabiħ il- Milied li għadda. My father bought me a nice toy last Christmas.
Audio file path:	_gugarell.mp3

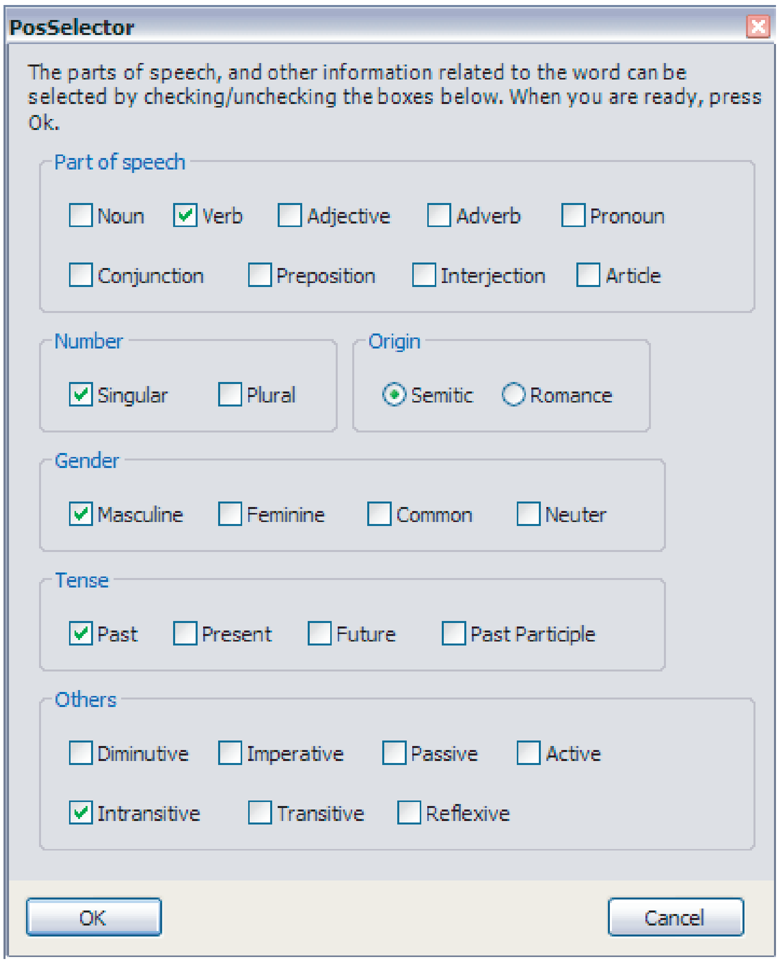
1

Login successful for user duncan@gmail.com

Figure C.2: Using the dictionary management tool.

they match a given prefix. This allows the user to view at most a set of words starting with the same letter. The records retrieved can be viewed either in grid mode or in record mode (Fig. C.2). This figure depicts an entry for the word *ġugarell* (toy), where relevant information such as the meaning and POS information is displayed. POS information can be entered by using the POS construction dialog (Fig. C.3).

The 'Audio file path' field is used to supply the word with spoken audio, like for


PosSelector

The parts of speech, and other information related to the word can be selected by checking/unchecking the boxes below. When you are ready, press Ok.

Part of speech: [ ] Noun [✓ Verb] [ ] Adjective [ ] Adverb [ ] Pronoun [ ] Conjunction [ ] Preposition [ ] Interjection [ ] Article

Number: [✓ Singular] [ ] Plural

Origin: [✓ Semitic] [ ] Romance

Gender: [✓ Masculine] [ ] Feminine [ ] Common [ ] Neuter

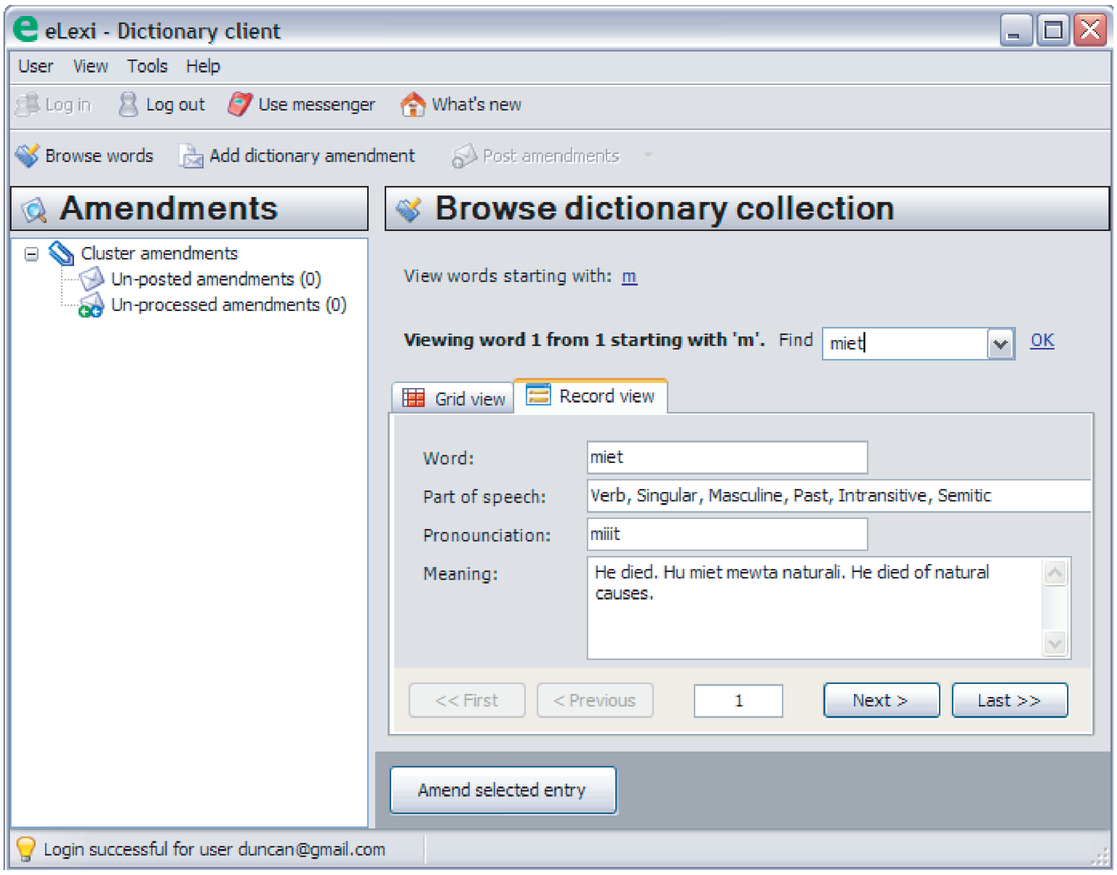
Tense: [✓ Past] [ ] Present [ ] Future [ ] Past Participle

Others: [ ] Diminutive [ ] Imperative [ ] Passive [ ] Active [✓ Intransitive] [ ] Transitive [ ] Reflexive

Figure C.3: The part of speech construction dialog.

example, the pronunciation. It must be made clear that as a value, only the file name must be supplied. Then, the respective file must be placed into `..\Website\audio` directory for the website to be able to reference it.

The management of amendments is very simple to carry out, and involves only pressing the *Accept* or *Reject* buttons. Deletion and creation of records in the tables underneath is automatically done for the user. Similar to the dictionary browsing facility, amendments can be viewed either in grid or in record mode. In addition to this, amendments for specified dates can be viewed by selecting the desired date from the provided calendar.



Viewing word 1 from 1 starting with 'm'. Find miet

Word:	miet
Part of speech:	Verb, Singular, Masculine, Past, Intransitive, Semitic
Pronounciation:	miiit
Meaning:	He died. Hu miet mewta naturali. He died of natural causes.

1

Login successful for user duncan@gmail.com

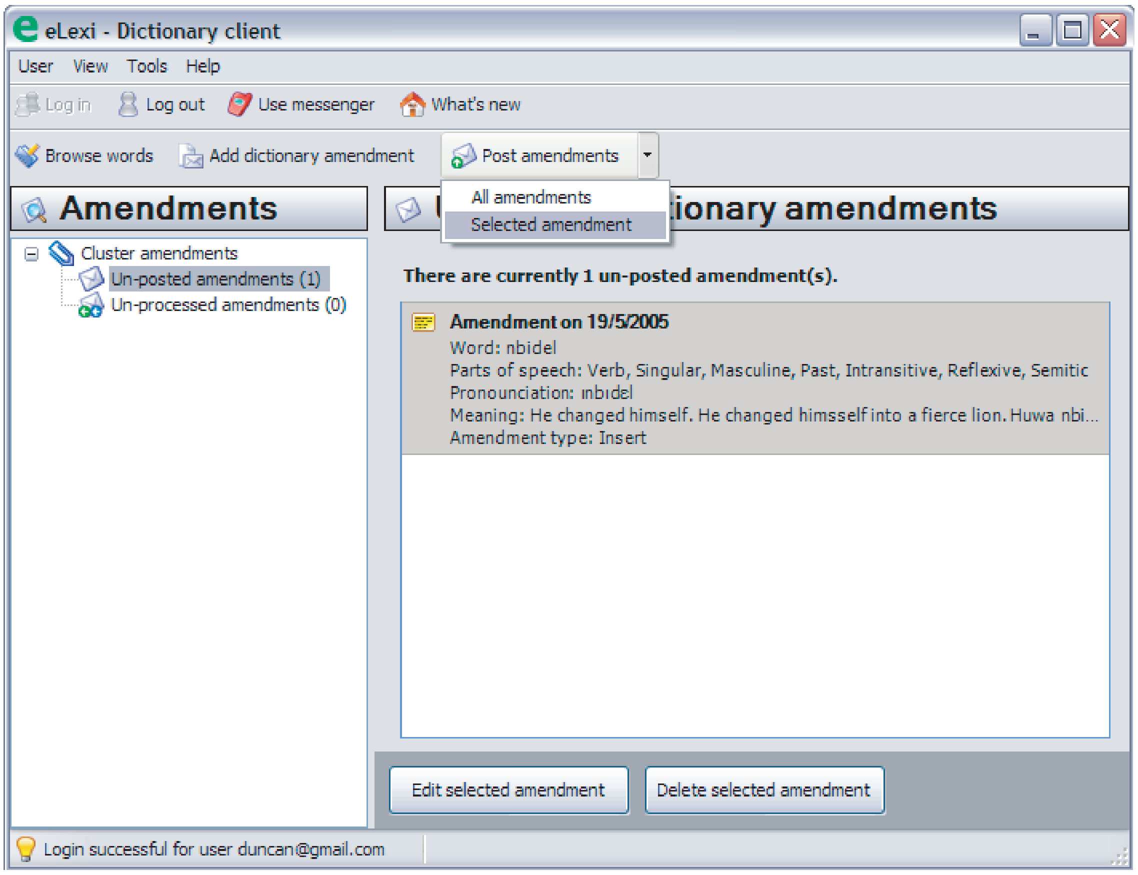
Figure C.4: The standard user dictionary tool.

C.2.2 The Standard User Plug-in

The standard user plug-in provides the opposite functionality that is provided by the administrator plug-in, in that the client is a producer of amendments and the administrator is a consumer — the buffers between these two are the amendment tables to which user amendments are posted. As was done in the previous subsection, only one tool will be considered, and again, the dictionary tool is discussed here (Fig. C.4).

As can be seen from Fig. C.4, amendments can be initiated by either selecting a dictionary entry or by posting a new amendment. In this current version of the plug-in, creating a new dictionary entry requires pressing the *Add dictionary amendment* button on the tool bar. In future versions, facilities for browsing the word list directly from the dictionary tool will be supplied.

The tree view on the left of Fig. C.4, provides the user with the option to view


eLexi - Dictionary client
User View Tools Help
Log in Log out Use messenger What's new
Browse words Add dictionary amendment Post amendments
All amendments
Selected amendment
Amendments
Cluster amendments
Un-posted amendments (1)
Un-processed amendments (0)
tionary amendments
Edit selected amendment
Delete selected amendment

There are currently 1 un-posted amendment(s).

Amendment on 19/5/2005
Word: nbidel
Parts of speech: Verb, Singular, Masculine, Past, Intransitive, Reflexive, Semitic
Pronounciation: ınbıdel
Meaning: He changed himself. He changed himsself into a fierce lion. Huwa nbi...
Amendment type: Insert

Login successful for user duncan@gmail.com

Figure C.5: Amendments stored locally.

both amendments which are stored locally or amendments which have been posted but not yet addressed by the administrator. Amendments which are stored locally can be posted all at once, or by specifically selecting the amendments to the posted. Should the user quit the application, un-posted amendments are automatically saved and loaded next time the application is loaded (Fig. C.5).

The user data files contained in `..\eLexi\AppData\user_name` where `user_name` is the currently logged user, are stored in XML file format, and can be conveniently copied, moved around and even used on other machines containing the eLexi software.

C.2.3 The Text Tools Plug-in

The text tools (Fig. C.6 on page 102) are general text processing tools which can be used by the administrator and the normal user alike. These tools are divided into three categories shown below:

1. *Word comparison*, allowing the analysis between pairs of words. Both pair-wise alignment as well as Levenshtein distance measurement are included here.

2. *Word statistics* providing the user with the facility to find out the frequency and probability of occurrence of words from a text file.

3. *Text analysis* tools which help the linguist process and make sense out of large amount of texts. The tools provide facilities such as:

   (a) Automatic prefix and suffix finding.

   (b) Automatic word segmentation using the list of affixes maintained in the database.

   (c) Automatic clustering of words from a text file, where clustering is based on either the root or stem of the words being processed.

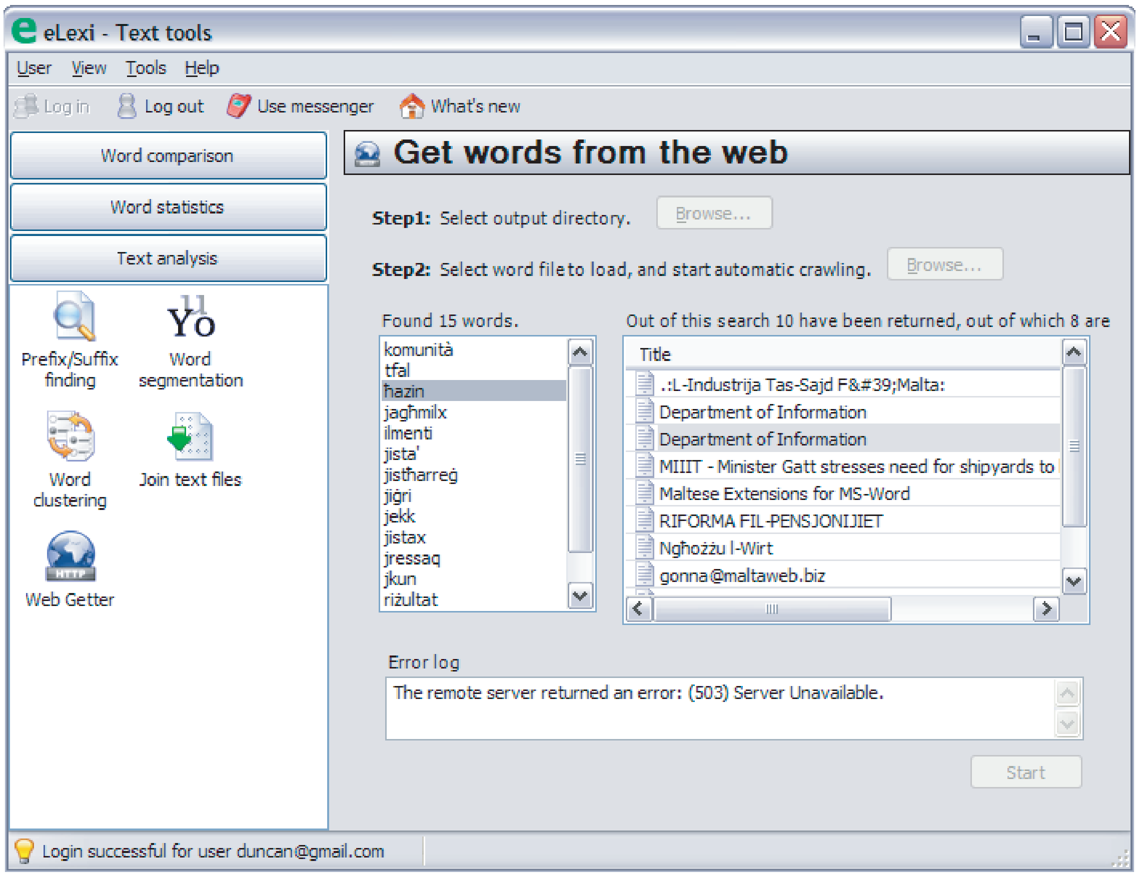
   (d) File joiner, allowing the user to form a word list from a number of files. This tool also supports rewrite rules which can be used to modify words or letters in the source text as specified by these rules. The changes are saved in the output text.

   (e) Web getter utility allowing the user to search the web for files containing a specified set of words. Once such files are found, these are automatically downloaded and parsed so that markup tags are removed.

Comments

This appendix provided a very brief and general overview on how to set up the system and use the plug-ins included. The user interface (UI) has been carefully crafted so that it is intuitive and easy to use. The design of the UI should not be taken lightly, since a carefully designed interface will immediately capture the user's attention without getting him confused in complex options and dialog boxes. In addition, a well designed UI eliminates the need of an extensive help system where very detailed descriptions are given.

The UI was designed partly by taking into account the new UI design model invented by Microsoft. The aim of the Inductive User Interface (IUI) model suggests new techniques by which the complexity can be reduced. Such techniques include



Get words from the web

Step1: Select output directory.

Step2: Select word file to load, and start automatic crawling.

Found 15 words.

komunità
tfal
hazin
jagħmilx
ilmenti
jista'
jistħarreġ
jiġri
jekk
jistax
jressaq
jkun
riżultat

Out of this search 10 have been returned, out of which 8 are

Title
.:L-Industrija Tas-Sajd F&#39;Malta:
Department of Information
Department of Information
MIIIT - Minister Gatt stresses need for shipyards to
Maltese Extensions for MS-Word
RIFORMA FIL-PENSJONIJIET
Ngħożżu l-Wirt
gonna@maltaweb.biz

Error log

The remote server returned an error: (503) Server Unavailable.

Login successful for user duncan@gmail.com

Figure C.6: The text tools available to the administrator and normal user.

providing limited options per screen, well defined tasks and using screens having clear titles and objectives[1].

As a last note, it must be kept in mind that this is a first release of the software, and therefore some bugs *are* bound to be present. However, by time, it is hoped that the system becomes more stable, thanks to the user comments and bug reports submitted in forums.

[1]An extensive report on IUI can be found at `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwui/html/iuiguidelines.asp`

Bibliography

Joseph Aquilina. *Maltese-English Dictionary.* Midsea Books, 1987-1990.

Jim Breen. *The EDICT Project*, 1999. Monash University.
URL `http://www.csse.monash.edu.au/~jwb/edict.html`.

Mons. Lawrenz Cachia. *Grammatika Ġdida tal- Malti.* Veriats Press, Żabbar, 1994.

Angelo Dalli. *Computational Lexicon for Maltese.* 2002. University of Malta, M.Sc. Thesis.

Angelo Dalli. *Data Representation Formats for Maltese.* 2001. University of Malta.
URL `http://mlex.cs.um.edu.mt/pub/datarp22.pdf`.

Anne de Roeck and Waleed Al-Fares. *A Morphologically Sensitive Clustering Algorithm for Identifying Arabic Roots.* 2000. In Proceedings of the 38 the ACL, Hong Kong.
URL `http://citeseer.ist.psu.edu/deroeck00morphologically.html`.

Hervé Déjean. *Morphemes as Necessary Concept for Structures Discovery from Untagged Corpora.* 1998. University of Caen-Basse Normandie.
URL `http://citeseer.ist.psu.edu/454401.html`.

Daniel Fasulo. An analysis of recent work on clustering algorithms. 1999.
URL `http://citeseer.ist.psu.edu/fasulo99analyis.html`.

Ana L. N. Fred and José M. N. Leitão. *A comparative Study of String Dissimilarity Measures in Structural Clustering.* 1998.
`URL: http://www.lx.it.pt/~fred/anawebit/articles/AFred_ICAPR98.pdf`.

John Goldsmith. *Unsupervised Learning of the Morphology of a Natural Language. Computer Linguistics*, 2001. University of Chicago.

URL `http://humanities.uchicago.edu/faculty/goldsmith/Linguistica2000/pdf/paper.pdf`.

Daniel Jurafsky and James H. Martin. *Speech and Language Processing.* Prentice Hall, international edition edition, 2000.

Dimitar Kazakov. *Unsupervised Learning of Naïve Morphology with Genetic Algorithms.* 1997. In W. Daelemans, A. van den Bosch, and A. Weijters, editors, Workshop Notes of the ECML/MLnet Workshop on Empirical Learning of Natural Language Processing Tasks, pages 105–112, Prague.
URL `http://citeseer.ist.psu.edu/kazakov97unsupervised.html`.

Paul Micallef and Mike Rosner. *Developing Language Resources for Maltese.* 2000. University of Malta. Proceedings of the Workshop on Resources for Minority Languages, Athens.
URL `http://citeseer.ist.psu.edu/596384.html`.

David W. Mount. *Bioinformatics: Sequence and Genome Analysis. Second Edition.* Cold Spring Harbor Laboratory Press, 2001.

Academy of Maltese. *Regoli tal- Kitba tal- Malti.* Klabb Kotba Mlatin, 1998.

M. Rosner, R. Fabri, J. Caruana, M. Loughraieb, M. Montebello, D. Galea, and G. Mangion. *Linguistic and Computational Aspects of Maltilex.* 2000. University of Malta. Proceedings of the Workshop on Resources for Minority Languages, Athens.
`http://mlex.cs.um.edu.mt/pub/athens2000.pdf`.

Mike Rosner, Joe Caruana, and Ray Fabri. *Maltilex: A Computational Lexicon for Maltese.* 1998. University of Malta.
URL `http://mlex.cs.um.edu.mt/pub/montreal98.pdf`.

David Sankoff and Joseph Kruskal. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* CSLI Publications, 1999.

Mike Scott. *Oxford WordSmith Tools Version 4.0: An integrated suite of programs for looking at how words behave in texts.* Oxford University Press.
URL `http://www.lexically.net/wordsmith`.

Utpal Sharma, Jugal Kalita, and Rajib Das. *Unsupervised Learning of Morphology for Building a Lexicon for a Highly Inflectional Language.* 2002. Tezpur University

and University of Colorado.
URL `http://citeseer.ist.psu.edu/596384.html`.