The Reverse Dictionary: A Lexical Similarity Measure using Semantic Networks

Jonathan Leigh
BSc Computing (Industry)
2005/6

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student)____________________
Summary

The reverse dictionary works in the opposite way to a normal dictionary. Whereas with a normal dictionary, a word would be submitted and the dictionary would return a set of other words which describe it; the reverse dictionary accepts a set of words which describe another word and the reverse dictionary returns the words(s) which best fit the description of the words supplied.

Utilising a custom built semantic network from the Longman Dictionary of Contemporary English, the Reverse Dictionary implements two approaches to measure lexical similarity.

The Reverse Dictionary was built from scratch using Python the project addresses issues relating the natural language processing and software engineering.

Also described in the project is a bespoke test for measuring the accuracy of a reverse dictionary known as the RD-Accuracy test. This test was developed for use with this project and with other reverse dictionaries.
Acknowledgements

With thanks to Eric Atwell, the project supervisor, for his guidance throughout the year; to Katja Markert for her input and suggestions; to the Support Staff for their technical expertise and assistance and to Kane for helping develop the idea.
Contents

1 Introduction 1

1.1 Aim ................................................. 1

1.2 Terminology ........................................ 1

1.2.1 Capitalisation .................................... 2

2 Background 3

2.1 Reverse Dictionaries .............................. 3

2.1.1 Definition ....................................... 3

2.1.2 Similar Projects ................................. 4

2.2 The LDOCE ......................................... 4

2.3 Semantic Networks ................................. 5

2.3.1 Definition ....................................... 5

2.3.2 Implementations ................................. 5

2.4 Measuring Lexical Similarity .................... 6

2.4.1 Using Semantic Networks ..................... 6
3 Design and Implementation

3.1 Design Considerations

3.1.1 LDOCE

3.1.2 Choice of a Relational DBMS

3.1.2.1 MySQL

3.1.3 The Stoplist

3.1.4 Lemmatizer

3.1.5 Python

3.1.6 Multiple Word Senses

3.2 Software Overview

3.3 Wordlynx

3.3.1 Overview

3.3.2 Database Schema

3.3.2.1 Tables

3.3.2.2 Indices

3.4 The Dictionary Reader

3.4.1 Overview

3.4.2 Design Considerations

3.4.3 Components
3.4.4 Execution Step 1: Parsing the LDOCE ........................................ 14
3.4.5 Execution Step 2: Loading Headwords and Links ...................... 15
3.5 The Inference Engine .............................................................. 16
  3.5.1 Overview ........................................................................... 16
  3.5.2 Design Considerations ...................................................... 16
  3.5.3 Algorithm ......................................................................... 16
    3.5.3.1 Original Algorithm ...................................................... 17
    3.5.3.2 Extended Algorithm .................................................... 17
  3.5.4 WordNet Similarity Measure ............................................... 17
  3.5.5 Components ...................................................................... 18
3.6 The User Interface ................................................................. 18
  3.6.1 Overview ........................................................................... 18
  3.6.2 Design Considerations ...................................................... 18
  3.6.3 The Graphical User Interface ............................................ 19
  3.6.4 The Text User Interface .................................................... 19
3.7 Other Software Packages ......................................................... 19
3.8 The RD-Accuracy Test ............................................................. 19
  3.8.1 Overview ........................................................................... 19
  3.8.2 Gathering Test Data .......................................................... 20
3.9 Project Methodology ............................................................... 20
Chapter 1

Introduction

1.1 Aim

As stated in the Mid-Project Report, the aim of this project is to create a working reverse dictionary. The aim of this report is to describe the design, implementation and results of the reverse dictionary as well as explaining the issues encountered during its completion and how they were addressed.

1.2 Terminology

This project report aims to be as clear as possible and any technical terms used are described fully. The following terms were used in the text but not defined and are explained here for clarity.

- **Regular Expression** A programming concept whereby a description of a string is used to find strings in a body of text.

- **WordNet** A semantic lexicon for the English language developed at Princeton University.
• **CGI** Common Gateway Interface, a standard for running programs from a website.

• **HTML** HyperText Markup Language, the standard document format for the World Wide Web.

• **Interpreted Language** A programming language which is execute from source code by an interpreter and is not compiled into object code.

### 1.2.1 Capitalisation

When referenced in this text, “Reverse Dictionary” refers to the product produced in this project and “reverse dictionary” refers to any reverse dictionary. Names of packages or classes in the produced software are capitalised, such as “User Interface” which refers specifically to the user interface for the “Reverse Dictionary”.
Chapter 2

Background

2.1 Reverse Dictionaries

2.1.1 Definition

Dictionaries are one of the most popular types of books available today. From the first years of primary school to the last years of life, any user of a language will consult a dictionary as part of day to day life. When using a dictionary, the user will look up a single word, known as the headword (which may consist of more than one word separated by spaces, known as a compound word) and will be given a definition, which can be defined as a set of descriptor words. Therefore, the function of a dictionary can be described as:

\[ \text{lookup}(h) \rightarrow d \]  

(2.1)

Where \( h \) is a headword and \( d \) is a set of descriptor words. A reverse dictionary functions by returning a headword given a set of descriptor words:

\[ \text{lookup}(d) \rightarrow h \]  

(2.2)
This could be considered as being similar to a thesaurus, but a thesaurus will return a set of headwords given a single descriptor word, whereas a reverse dictionary will return one or more headwords given a set of one or more descriptor words.

2.1.2 Similar Projects

Although some implementations of reverse dictionaries exist, it seems that little academic research has been carried out in the field. This does not mean to say that there is nothing in academia which is related to this project. One successful implementation was created by OneLook and is available online at http://www.onelook.com/reverse-dictionary.shtml. By OneLook’s own admission, the system was created using “a motley assortment of statistical language processing hacks.” [3]. In contrast, this system was created using a set of well documented, academic methods.

One should be wary of reverse dictionaries which are, in fact backwards dictionaries. A backwards dictionary list words not in “front-to-back” alphabetical order, but ordered in a “back-to-front” fashion, that is listed in order by which the word ends. An example of such a work is Lehnert’s reverse dictionary[11].

2.2 The LDOCE

Background research has revealed that much work has been done using the LDOCE (Longman Dictionary of Contemporary English). This dictionary was released in electronic format on tape by Longman in 1978 [13] and was one of the first dictionaries of the English language available in machine readable format. As a result many other projects in the field of computational linguistics have used the LDOCE as a data source. The MRD (Machine Readable Dictionary) was “LISPified” [4] (p.47) when distributed as it was intended for use by NLP and other AI systems. This is fitting as at the time, Lisp was the primary high-level language used for Artificial Intelligence development and research.

The LDOCE is also particularly useful for NLP work as the definitions of each headword are drawn from the Longman Defining Vocabulary (henceforth LDV), which consists of 2851 words and is considered to be the basic set of words which makeup the English language. Any other word can be described using
The use of the LDOCE as a database is discussed by Broguraev [4] (pp. 52-61) but the use of a relational database to hold a semantic network was not considered. Demetriou[6] also used the LDOCE as a semantic knowledge base, with the emphasis placed on speech transcription.

The LDOCE was supplied on tape in a “LISPified” format with no documentation. Each entry was split into a number of numerically classified fields, the purpose of many was not obvious. A large amount of research was required to discover the purposes of these fields and decode the information contained within in order that a dictionary parser be designed. [4] (pp. 269-271) gave decodes for the syntactic codes and subject fields, but a full list of decodes was not listed. Drakos[7] gave a more detailed list of the decodes for the subject and semantic codes.

Finally, a full description of all unknown fields was found in the University of Leeds School of Computing [1]. The information found in this document made it possible to complete the design of a dictionary parser.

### 2.3 Semantic Networks

#### 2.3.1 Definition

A semantic network is a method for indicating the relationships between different words based on their meaning. For example; car, lorry and train are all semantically linked as they describe vehicles and therefore would be close in a given semantic network. The term was first coined by Quillian[18] but the concept can be traced back to the ancient Greek philosopher Aristotle [9].

#### 2.3.2 Implementations

One of the most well known implementations of a semantic network is WordNet, developed at Princeton University. It groups words into synonym sets (synsets) which are connected to other, related synsets to form a large semantic network [15].
Kozima & Furugori [8] also developed a semantic network known as Paradigme which was also built from the LDOCE. Another similarity to this project was that Kozima and Furugori used their semantic network to measure similarity between words.

Ho & Fairon [9] proposed a novel method for measuring lexical similarity using semantic networks which will be discussed further in section *Measuring Lexical Similarity*.

### 2.4 Measuring Lexical Similarity

There are a number of recognised methods for computationally determining the similarity between two given words. One of the original methods was developed by Michael Lesk in his paper *How to tell a Pine Cone from an Ice Cream Cone* [12] and uses a simple but effective method of comparing crossover in the definitions derived from the Oxford English Dictionary. Leacock and Chodorowi [10] used a different approach, comparing the lengths of paths between words in a given *is-a* hierarchy, implemented by Pedersen et. al. [16] using WordNet. Other methods are discussed below.

#### 2.4.1 Using Semantic Networks

Here there are two works which are highlighted because they propose methods for measuring lexical similarity using semantic networks; they are Kozima & Furugori [8] and Ho & Cedrick [9]. As mentioned previously, Kozima & Furugori developed a scaled-down semantic network (Paradigme) while Ho & Cedrick based their semantic network on the entirety of the Webster dictionary.

The two approaches also differed in the way semantic networks were compared to each other. Ho & Cedrick used at the quantity of information exchanged between two networks:

\[
sim_{QIE}(A,B) = f(mfi\omega(A,B),mf\omega(B,A))
\]  

(2.3)

Kozima & Furugori used a simulation of spread of activity through Paradigme, with each node calculating its own activation level using

\[
v(T + 1) = \phi(R(T),R(T),e(T))
\]  

(2.4)
2.4.2 Applications

There appears to be a limited number of applications for measuring lexical similarity. Word sense disambiguation is certainly the primary use as demonstrated by Patwardhan et. al. [15] who implemented the Adapted Lesk Algorithm [2] as a method for word sense disambiguation. Ho & Cedrick [9] developed a synonym extractor (thesaurus) using lexical similarity and semantic networks. The application described in this report appears to be a novel use for these well developed techniques.
Chapter 3

Design and Implementation

3.1 Design Considerations

3.1.1 LDOCE

The LDOCE was chosen as the primary datasource for this project. This design choice was made for a number of reasons. Firstly, the LDOCE was easily available in a machine readable format as it had already been acquired by the university. This meant that it was not necessary to purchase or gain permission to use another lexicon as well as aiding in the reading and parsing of the dictionary.

Secondly, the definitions contained in the LDOCE are formed from the Longman Defining Vocabulary[13] which is a subset of the English language consisting of 2197 words and is considered the smallest set of words which can be used to describe any other word in the dictionary. As a result, all words contained in a definition will also be found as headwords inside the LDOCE, allowing a complete, although slightly limited semantic network to be created.
3.1.2 Choice of a Relational DBMS

It would have been very inefficient to re-generate the semantic network every time it was to be queried so non-volatile storage of data was a requirement. Also this storage would need to efficiently store both string data and data pertaining to the relationships between instances of this string data. The data store needed to be able to quickly identify and return any piece of data given its identifier, and identifiers needed to be created on-the-fly and maintained. Finally, the data store would need to be accessible by more than one software architecture, possibly written in different languages and running on different machines. For example, the semantic network creation software may run on an undergraduate workstation while the inference engine runs on a web-facing server.

Given these prerequisites, a relational database management system (henceforth rDBMS) was chosen. A rDBMS also has the advantages of being a mature concept in wide use throughout the industry. Further advantages of using a rDBMS is that the software would be installed and maintained centrally, and the data backed-up regularly and securely. Examples of rDBMSs include Microsoft SQL Server, MySQL and Postgress.

3.1.2.1 MySQL

The rDBMS which was used for this project is MySQL (version 4.1.15). MySQL is a popular and advanced open source relational database which runs on a number of platforms. Although other rDBMSs could have been used, MySQL was chosen because it was the best supported DBMS in the school and the author had previous experience using MySQL and a SQL code in general.

3.1.3 The Stoplist

The stoplist is a list of words which would be ignored in an input stream by the system. The input stream is a set of words and may originate from a user querying the dictionary or from the dictionary reader as it parses the input data. It fulfils two functions; firstly to prevent querying taking place on common words and secondly to remove formatting data from the dictionary.
When submitting a query to the dictionary common connective or prepositional words (such as “and”, “the”, etc.) will likely be included. These words do not help define the semantics of the phrase submitted by the user (for the purposes of the similarity measures used by this project), but would increase the complexity of the search of the semantic network, reducing responsiveness and therefore are considered undesirable. For this, a standard stoplist was used.

The machine readable LDOCE originated from the electronic version which was sent to the printers and therefore contains some typesetting and formatting information which is in the form of sets of capitalised letters three or four characters long. These formatting codes were not identified as words not part of the definition by the regular expression used in parsing and therefore were added to the stoplist. A full list of these codes was not available and therefore were added to the stoplist by analysing the execution log after each run.

The final stoplist contained 60 words.

### 3.1.4 Lemmatizer

A lemmatizer reduces a word to its most basic form, that is present tense singular. It is similar to a stemmer but instead of returning a stem which may or may not be a real word, it always returns a word which can be found as a headword in a dictionary. This table shows the difference in output from a stemmer and lemmatizer given the same input:

<table>
<thead>
<tr>
<th>Original Word</th>
<th>Stemmed Word</th>
<th>Lemmatized Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>neighbours</td>
<td>neighbour</td>
<td>neighbour</td>
</tr>
<tr>
<td>were</td>
<td>were</td>
<td>be</td>
</tr>
<tr>
<td>computational</td>
<td>comput</td>
<td>computational</td>
</tr>
</tbody>
</table>

Table 3.1: Outputs from The Porter Stemmer[17] and the RASP Lemmatizer[5]

The lemmatizer used in this project was taken from the RASP toolkit which utilises 1400 finite state rules and has an error rate of 0.07%[5]. This lemmatizer was chosen for its high accuracy, fast response time and availability (it has been installed as part of the natural language processing toolkit on csunix and available for undergraduate use).
3.1.5 Python

The software to build and query the semantic network was built in Python. Python was chosen for a few reasons, but portability was the primary consideration. In the future, the project may be ported to different servers running on different platforms, so software capable of running on different operating systems was a fundamental requirement. Also it is relatively straightforward to create CGI applications in Python, a requirement for the HTML based user interface.

Python is also considered to be well suited for natural language processing in the academic environment[14], supports object orientated code and is easy to learn, write and read as it is an interpreted language with a clear syntax.

3.1.6 Multiple Word Senses

Often an entry in the LDOCE will have more than one word sense. These may be split over different entries such as is the case for the word bank or the multiple senses may be combined into one entry, such as in the word basin.

In both cases the Reverse Dictionary will consider all definitions to be part of the one atomic headword. Also, definitions belonging to an entry in the LDOCE whose headword can be lemmatized to another headword will be grouped together to the lemmatized headword. Similarly, definitions for different parts (noun, verb etc.) of speech will be added to one headword.

The reason for this concatenation of multiple senses is to reduce duplicates appearing in a result set and to reduce the number of links to be traversed in the semantic network when searching.

3.2 Software Overview

The software for The Reverse Dictionary is split into three packages; the Dictionary Reader, the Inference Engine, the User Interface and the Knowledge Base (Wordlynx). These packages were built specifically for this project. Some other packages were used which are described in the section Other
Software Packages. Figure 3.1, below, shows the major software modules and the interaction between them.

![Diagram showing software modules and their interaction. The arrows represent the direction of the flow of data](image)

3.3 Wordlynx

Wordlynx is the name given to the Knowledge Base, a semantic network stored in a relational database created exclusively for use in this project.

3.3.1 Overview

Wordlynx stores the semantic network by maintaining a list of headwords and a list of links between the headwords. Each headword is assigned a unique identifier (an integer number generated by an autoincrement field). The relationships between headwords are stored as a set of two headword identifiers, one of the parent word and one of the child word. This provides a set of equally weighted, directional paths between nodes in the network. Each headword may be connected to zero or more other headwords.

A child word is linked to a parent word if it appears in the definition of that parent word. Although a
distinction between child words and parent words is made, parent words may also be children of other words.

### 3.3.2 Database Schema

#### 3.3.2.1 Tables

There are two tables in the database, they are:

- **Words** - The headwords and their unique identifiers
- **Links** - The set of links between headwords.

The schema of these tables is as follows (primary keys marked with an asterisk):

<table>
<thead>
<tr>
<th>Table</th>
<th>Column</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Words</td>
<td>word_refno*</td>
<td>int</td>
<td>Autoincremented unique word identifier</td>
</tr>
<tr>
<td></td>
<td>word</td>
<td>varchar(50)</td>
<td>The word</td>
</tr>
<tr>
<td></td>
<td>type</td>
<td>char(1)</td>
<td>Unused, included for forward compatibility</td>
</tr>
<tr>
<td>Links</td>
<td>link_refno*</td>
<td>int</td>
<td>Autoincremented unique link identifier</td>
</tr>
<tr>
<td></td>
<td>parent_refno</td>
<td>int</td>
<td>Pointer to the parent word</td>
</tr>
<tr>
<td></td>
<td>child_refno</td>
<td>int</td>
<td>Pointer to the child word</td>
</tr>
</tbody>
</table>

Table 3.2: Schema of Wordlynx

#### 3.3.2.2 Indices

A single index was placed on the `word` column of the `words` table. This index improves performance of the Inference Engine and Dictionary Reader by making it easier for the DBMS to identify records. A non-clustered B+Tree was used.
3.4 The Dictionary Reader

3.4.1 Overview

The Dictionary Reader is the software which builds the semantic network from the LDOCE and loads it into Wordlynx. It is the largest of the three software packages and takes about 6 hours to run from beginning to end.

3.4.2 Design Considerations

The Dictionary Reader was designed with a number of factors in mind, primarily portability and extensibility. The package was designed to be used in a “plug and play” environment whereby the database and data source (dictionary) could be changed with limited impact. This was for two reasons; to allow the project to be run on other servers, running on different platforms with different databases available and to allow for different dictionaries to be read into the database to improve accuracy of the reverse dictionary.

3.4.3 Components

There are a number of components in the program, encapsulated as Python classes. They are listed and described in the table 3.3 below.

3.4.4 Execution Step 1: Parsing the LDOCE

The LDOCE Parser has the task of extracting all useful data from the “LISPified” machine readable version of the LDOCE in preparation for use by other parts of the program. It reads each line of the LDOCE in turn and identifies what data is contained in the line by inspecting the first few characters.

The start of an entry is identified by a character sequence and a regular expression extracts the headword and the program creates a LDOCEEntry object to store the data in.
If the line is the start of or forms part of a definition of the entry (there may be more than one definition for different senses), a regular expression extracts the descriptor words (words which form the definition) and add them to the LDOCEEntry object.

If the line marks the end of an entry then the LDOCEEntry is "pickled". Pickling is the Python name for serialisation, that is converting an Object to a stream of binary data which can be saved to disk for later retrieval, in the same way a vegetable would be pickled for eating at a later date. The entries are pickled by the parser so that they can be read by the Headword Reader and the Link Loader at a later time. This functionality, although heavily consuming disk space means that the parser and loaders can be run at different times, and also if the process is interrupted, it could be resumed from the last entry completed successfully. As the entire process of parsing and loading is lengthy this is a great advantage.

### 3.4.5 Execution Step 2: Loading Headwords and Links

The LDOCEHeadwordReader loads the headwords extracted by the parser into the database. As with all modules, all interaction with the database is carried out by the Database Interface. It generates a list of parsed headwords by a system call to get the filenames of all files in the directory holding the pickles. Each file is "unpickled" into an LDOCEEntry object whose headword is inserted into the database by a single function call to the Database Interface.

The Link Loader is a little more complex. Once again the list of headwords is retrieved from the directory listing of the pickles directory. Each word is unpickled to a LDOCEEntry and the database is searched to find the identifier of the word. The identifier is required to insert the links as it will be used as the `parent_refno` column in table `links`. The identifier for each descriptor word, after being lemmatized, is found from the database and a link record added, once again by a single call to the Database Interface.
3.5 The Inference Engine

3.5.1 Overview

The Inference Engine performs the searches against the Reverse Dictionary. It accepts a set of descriptor words and returns an ordered list of headwords. The input is encapsulated as an instance of a `WordSet` class and returns its results as an instance of a `ResultSet` class.

3.5.2 Design Considerations

The Inference Engine was designed to work completely independently of the Dictionary Reader, meaning that the source of the semantic network is irrelevant to the workings of the Inference Engine. It does not directly access the database so any Semantic Network could be used, provided a compatible database interface is available.

The Inference Engine is tied more closely to the User Interface, but not to the extent that it is reliant on one type of user interface. There are currently two user interfaces which work with the Inference Engine, one HTML based, the other text based. They work together seamlessly.

This modular design affords for maximum portability and extensibility of the project.

3.5.3 Algorithm

The algorithm which the Inference Engine uses has been extended since the original design; both the original design and the extended algorithm are described here.

Before continuing this discourse the `Voter` class must be introduced as it is key to the Inference Engine’s algorithm. The Voter class is a data structure similar to a hashtable (a list of key-value pairs) which accepts weighted votes for a headword. It publishes a method called `vote` which accepts a string and a floating point number (float). The string is the word to vote for and the float is the weight of the vote (from zero to one). If `vote` is called with a word which has already been voted for, the weight of the vote is added to the currently stored vote. Voter will return an ordered list of all words voted for and
their weights. Henceforth, if a headword is "voted for", this is achieved by a call to vote of the currently
maintained Voter object.

### 3.5.3.1 Original Algorithm

The algorithm is a multi-level search, with the number of levels configurable via the global settings file.
First a Voter is initialised, and the descriptor word set is prepared by removing duplicates and passing it
through the stoplist. Each descriptor word provided as input is looked up in the semantic network and
their parent words are voted for. The headwords voted for by one level are passed as the input to the
next level. The weight of the vote for the first level is 1.0. The weight of the vote given for subsequent
levels is $n = l \div 2$. Where $l$ is the current level. Once the search is complete at all levels, the result set
is pruned to remove results with low weighted votes. This reduces the number of results returned to
the user, improving the perception of accuracy of the system by reducing the number of loosely-related
results and therefore improving the user’s experience.

### 3.5.3.2 Extended Algorithm

The currently implemented algorithm is extended beyond the minimum requirements. The Inference
Engine uses WordNet to improve the accuracy of the results. Each parent word returned by the search of
the original algorithm is compared to the child word (descriptor word) to return a scalar value indicating
their similarity. Details of how this is achieved is covered in the section WordNet Similarity Measure.

### 3.5.4 WordNet Similarity Measure

The WordNet similarity measure relies on the fact that WordNet stores words as unary trees mapping the
word’s categories at numerous levels. An example tree for the word “cat” is: entity → physicalentity →
object → livingthing → organism → animal → chordate → vertebrate → mammal → placental →
carnivore → feline → cat. The trees from two words are compared and the point at which they intersect
is the measure of similarity. For example, if “cat” were compared to “dog” the trees would intersect at
“carnivore”, just 2 branches from the top of the tree indicating a high similarity; while the word “sky”
would intersect with “cat” at “physical entity”, at the bottom of the tree, indicating a low similarity. The similarity value is calculated by the equation

\[ s = \text{meet}\_\text{position} \div \text{tree}\_\text{length} \]  

Where \( \text{meet}_\text{position} \) is an integer number indicating the distance down the headword’s tree where the descriptor word’s tree intersects it.

The extended algorithm adds this similarity to the vote to give additional weighting to it.

### 3.5.5 Components

The program is made of numerous components, all written in Python. Other components are used but they have already been described in the Dictionary Reader section. Table 3.4 below describes these components.

### 3.6 The User Interface

#### 3.6.1 Overview

There are two user interfaces in the system; one is an interactive, text based program and the other is a HTML based graphical interface. Both interfaces are quite simple, yet perform the task as required. The HTML interface uses a CGI script, written in Python to parse and send the descriptor words to the Inference Engine and to format and present the results.

#### 3.6.2 Design Considerations

Like the other modules in the project, the User Interface was designed to be independent of the Inference Engine. This is achieved by using custom classes to transfer data to and receive results from the Inference Engine. If the type or format of the User Interface is altered, no changes are required to be made to the Inference Engine or Dictionary Reader.
3.6.3 The Graphical User Interface

The GUI is written in HTML and includes a text box where the user can enter the descriptor words. When the submit button is clicked, the descriptor words are posted to a CGI script written in Python. The CGI script parses the descriptor words and calls the Inference Engine to perform the search. Once the search is complete the results are retrieved from the Inference Engine and formatted as a HTML table and displayed on the user’s web browser, ordered by similarity.

3.6.4 The Text User Interface

This user interface runs on a UNIX shell or Windows command line. It first asks users to enter descriptor words one at a time and allows compound words to be entered. Once all descriptor words have been entered, the Inference Engine is called to perform the search. The progress of the search is shown using a progress bar printed out on to the terminal. The results are then printed out ordered by similarity.

3.7 Other Software Packages

Some other packages were used in the project, table 3.5 below acknowledges all of them.

3.8 The RD-Accuracy Test

3.8.1 Overview

The RD-Accuracy Test is a test designed specifically for this project which aims to measure the accuracy of any reverse dictionary. It is used to determine the success of the project both by determining how well it performs and if it performs better than other reverse dictionaries.

The RD-Accuracy Test is a scenario based test requiring user input. Test data was collected which consisted of headwords and up to six descriptor words which are related to the headword. The descriptor words are entered into the reverse dictionary and the (ordered) resulting headwords analysed. The
location of the original headword in the ordered list of results determines the score for that test. The test is repeated for all test data and the scores awarded for each test averaged. The overall score for that reverse dictionary can then be recorded.

The score for each test is determined by subtracting the position from 100. Positions over 100 are ignored (awarded zero) and headwords which did not appear in the results at all are awarded zero. The test is discussed in more detail in the chapter *Results and Evaluation*.

### 3.8.2 Gathering Test Data

Volunteers were asked to “think of a word” and then provide up to six other words which “describe or are semantically linked” to that word. Volunteers were drawn from friends, family and other students. They were all English speaking. For privacy reasons, no logs were kept apart from the data inputted, none of which is personally identifiable.

Submissions were made via a website which consisted of a HTML page which provided a brief description of the test which sent the data to a CGI script which recorded the results. There results were manually edited to remove unsuitable and obscene entries.

80 entries were collected in total and are provided in Appendix C. The website used to collect the data can be viewed here [http://www.jonnyleigh.co.uk/submitWords.html](http://www.jonnyleigh.co.uk/submitWords.html).

### 3.9 Project Methodology

To aid the success of a large software project, a suitable project methodology must be selected and adhered to. For this project, the waterfall model was chosen. Although considered inflexible and outdated by some, it was felt that this was the best methodology for this project.

The waterfall model was considered suitable for the following reasons. Firstly its clearly defined phases allow for more precise budgeting of time and lean towards a fixed deadline as was required for this project. Unlike the iterative waterfall or RAD (Rapid Application Development) models, the waterfall model lacks flexibility in the face of latent design flaws discovered in later phases and other unforeseen
problems, a "buffer period" was built into the project plan to allow for any required changes to be made. Another way that unforeseen issues and requirements were accommodated was by using elements of Rapid Application Development methodology. For example the requirement for separate connective classes for input and output data from the Inference Engine was not foreseen, it was assumed that a single connecting class would be sufficient. When this discovery was made, development of the Inference Engine was well underway and rather than break from the planned development to produce a new design, the new class was quickly coded, with the requirements fresh in mind. Although this is not entirely suitable for a large project being worked on by a team, it is acceptable for a project of this size, with only one developer.

Secondly, the waterfall model places emphasis on the production of design documentation which drives the software development stages as well as aiding future development. A good design document helps prevent issues occurring in later phases of the project. For example, the requirement for a text-based user interface was not initially planned for, but was required due to an impending demonstration when the graphical user interface was not yet ready. Due to the modular design initiated in the early phases, it was relatively trivial to produce another user interface which worked surprisingly well when run in parallel with the pre-planned, graphical user interface.

Overall the waterfall model is a mature methodology and is well suited for this project, even in face of newer methodologies such as The RUP and RAD.

The project was planned using a schedule (built using a spreadsheet). The plan was quite broad, covering phases such as “Design RD-Accuracy test” or “Background research” and included mandatory milestones (mid-project report / final deadline) and personal milestones such as "release product v1" and "complete testing”. Overall the plan was kept to, although the release and tests were carried out before the testing of competitors, contrary to the original plan. This was because it was more efficient to test the competitors and the product at the same time.
3.10 Tools and Practices

To ensure smooth development and testing, a number of tools were used. Tools ranged from spreadsheets to scripts. They are briefly described in this section, although not considered deliverables. Also described here are some practises kept to during the project. These practises were in order to aid the development and maintenance of the software.

An execution log of the Dictionary Reader was kept which tracked the version numbers of the components of the Dictionary Reader and any issues which occurred during each run. Four runs of the Dictionary Reader were made before the database was usable by the Inference Engine.

Executing the Dictionary Reader was a complex task involving numerous software components and file operations which took nearly a full day to complete. Although only four runs of the Dictionary Reader were made, a number of dry runs were completed during development to aid in debugging. Therefore a shell script to perform the file operations (backup execution log, delete old pickles) which then called a Python script to parse the dictionary, load the headwords and load the links.

Building of the User Interface consisted of a HTML file and a CGI file, both of which needed to be deployed in separate directories, with different permissions set. To help with this the master files were kept together in one, off-line directory and a shell script was written which copied the files to the web-facing directories and set the required permissions.

A number of automation scripts were written for testing. Once the test data had been collected it could be parsed into instances of the WordSet class. One Python script was written which ran all the test data through the Reverse Dictionary, storing the results as pickled ResultSet. Another was written which sent the data to the Onelook Reverse Dictionary using the sockets API, storing the resulting HTML data. Once all the tests had been carried out, the data could be analysed, using a spreadsheet. For final testing, the scripts were not used as explained in a later section.

All software modules had a header block with version number, date and a description of the module. Many methods inside the modules also had a header block detailing the actions, parameters and return values of the method.
The Object Orientated software model was used in all Python modules, except for the most basic scripts. Python, like Delphi and VB can be coded either using Objects or in the traditional procedural style, but a consistent style was deemed necessary. All program settings were stored in a global settings file which was accessed through a class called SettingsReader, allowing all software components to be controlled centrally, making for easy fine tuning of the application. “Magic Numbers” and “Magic Strings”, that is any significant constant in the software, were controlled using the global settings file.

Similarly, all program output was handled by one class called Output. Whenever status information would be outputted to the console, it is sent via Output which would print to the console if verbose mode was on and save it to a log file otherwise. This meant that output can be limited during production use by will be fully visible to the developer during development and testing.

An attempt was made throughout the development to adhere to generally accepted good coding practises. Global variables were minimised to prevent ambiguity and sensible variable names were used in most cases. Python enforces proper indentation which aids readability of the source code.

### 3.11 Difficulties Encountered

A number of difficulties were encountered during the development of the software. The first was when building the database as a MS SQL Server database was originally planned and designed for. However, it was not possible to gain access to a MS SQL Server database so MySQL was used instead. Due to the simplicity of the database design and similarities in the SQL code for both platforms, it was trivial to convert to a MySQL database. As a result of the modular design of the software, updating the Database Interface was all that was required to accommodate this change, having no impact on other parts of the software.

Parsing of the LDOCE raised a number of issues. Firstly typesetting and formatting codes were being imported along with the descriptor words, which caused the Link Loader to fail to find headwords, disjointing the semantic network. This was due to limitations of the Regular Expression used. The stoplist was updated to remove the superfluuous codes. Secondly was a disk space limitation. The storage of the pickles required a large amount of disk space, more than was available under the standard
disk quota for undergraduates. By sending a request to Support, the disk quota was increased to allow
room for the pickles. This problem was not mitigated in the design phase as the size of the pickles was
unknown until development had started.

Running the Dictionary Reader is a lengthy process, taking up to a full day to complete. The program
was run on a central server called cslin-gps which was connected to via a secure shell. Cslin-gps
enforces a maximum processor time of 30 minutes. Any process which has been running for longer
than 30 minutes is terminated automatically to prevent processes from entering an infinite loop. The
Link Loader was terminated before completion as a result of this policy. It was not known at the time
that it was possible to override this policy so to mitigate this problem, the Link Loader was updated to
accept a parameter limiting the range and number of words to process. A shell script was written which
iteratively ran the Link Loader for each letter of the alphabet which successfully prevented the “CPU
time exceeded” error message.

Interfacing with WordNet presented some interesting technical difficulties. The API provided by Word-
Net is written in Perl which cannot be directly interfaced by Python. A package for running Perl com-
mands in Python does exist (PyPerl) which is no longer maintained as is riddled with bugs. Compilation
of the library took half a day, but implementation errors made it unusable. Eventually a Python interface
to WordNet was found and used successfully.

These difficulties highlight some of the challenges of the project. Due to buffer periods being built into
the project plan, development was kept on schedule in the face of these problems.

3.12 The Final Product

The final product can be viewed at http://wwwdev.comp.leeds.ac.uk/scs2jjl/
<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache.py</td>
<td>A memory cache for word / identifier pairs utilising a LFU replacement algorithm. Reduces number of database queries made.</td>
</tr>
<tr>
<td>DatabaseInterface.py</td>
<td>General interface to the database, configured to use MySQL.</td>
</tr>
<tr>
<td>LDOCEEntry.py</td>
<td>A single entry (headword and definition) from the LDOCE</td>
</tr>
<tr>
<td>LDOCEHeadwordReader.py</td>
<td>Loads headwords, already parsed from the LDOCE into the database</td>
</tr>
<tr>
<td>LDOCELinkLoader.py</td>
<td>Reads definitions from pickled LDOCEEntries and loads the links into the database</td>
</tr>
<tr>
<td>LDOCEParser.py</td>
<td>Reads the LDOCE from its file and store it as a list of LDOCEEntry objects.</td>
</tr>
<tr>
<td>Lemmatizer.py</td>
<td>Interface to the RASP lemmatizer.</td>
</tr>
<tr>
<td>loadHeadwords.py</td>
<td>Non-Dictionary specific interface to load headwords into the database</td>
</tr>
<tr>
<td>loadLinks.py</td>
<td>Non-Dictionary specific interface to load links into the database</td>
</tr>
<tr>
<td>output.py</td>
<td>General logging class for outputting status data to log file and console</td>
</tr>
<tr>
<td>parse.py</td>
<td>Non-Dictionary specific interface for parsing a dictionary</td>
</tr>
<tr>
<td>ProgressBar.py</td>
<td>Displays a status bar on the screen</td>
</tr>
<tr>
<td>SettingsReader.py</td>
<td>Python interface to the global settings file</td>
</tr>
<tr>
<td>StopList.py</td>
<td>Interface to the stoplist</td>
</tr>
</tbody>
</table>

Table 3.3: Components of the Dictionary Reader
<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>InferenceEngine.py</td>
<td>Inference Engine main program</td>
</tr>
<tr>
<td>ResultSet.py</td>
<td>Ordered list of headwords and their cumulative vote weights</td>
</tr>
<tr>
<td>Voter.py</td>
<td>Class to store weighted votes for headwords</td>
</tr>
<tr>
<td>wntools.py</td>
<td>WordNet extended tools</td>
</tr>
<tr>
<td>wntree.py</td>
<td>Class to store word tree from WordNet</td>
</tr>
<tr>
<td>wordnet.py</td>
<td>Python interface to WordNet</td>
</tr>
<tr>
<td>WordNetSimilarity.py</td>
<td>Calculates similarity of two words using WordNet</td>
</tr>
<tr>
<td>WordSet.py</td>
<td>A list of words, used as input to the Inference Engine</td>
</tr>
</tbody>
</table>

Table 3.4: Components of the Inference Engine

<table>
<thead>
<tr>
<th>Package</th>
<th>Author</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQLPython</td>
<td>Andy Dustman</td>
<td>Python interface to MySQL database</td>
</tr>
<tr>
<td>RASP</td>
<td>Kevin Humphreys, John Carroll and Guido Minnen</td>
<td>Parsing and morphological tools. Lemmatizer used</td>
</tr>
<tr>
<td>pyWordNet</td>
<td>Oliver Steele</td>
<td>Python interface to WordNet</td>
</tr>
</tbody>
</table>

Table 3.5: Other Software Packages
Chapter 4

Results and Evaluation

4.1 Evaluation Criteria

4.1.1 Overview

The Reverse Dictionary is evaluated on its accuracy as this ultimately determines its value to a user. The accuracy of a reverse dictionary is a subjective term and a full definition of this concept is discussed fully in this section.

4.1.2 Accuracy

When a user accesses the Reverse Dictionary to find a word to describe a concept or to find synonyms they will want their headword to appear in the result list. As the results are sorted by similarity, the order of the results is non-trivial. A successful search will display the headword in the result set, but a more successful search will display it at or near the top of the list. A large result set will take time to read
and so to make the user experience as pleasant as possible, they should be able to find their headword quickly. A reverse dictionary will often return many results which when viewed by a human will appear as unrelated. For example, given the test set \((\text{monkey}, \text{animal}, \text{gorilla}, \text{ape}, \text{hairy})\) the results returned are \((\text{chimpanzee}, \text{peccary}, \text{goat}, \text{eyelash}, \text{gibbon}, \text{anthropoid}, \text{anthropoid ape}, \text{yeti})\). The words \text{goat} and \text{eyelash} are clearly poorly related to the headword \text{monkey}, but it is possible to understand why the Reverse Dictionary returned these results. If the user’s headword is buried below 100 other seemingly unrelated words, the user experience will be diminished. For these reasons, the location of the headword within the result set, is considered a good measure of accuracy.

Accuracy is a useful evaluation criteria because as well as having a direct effect on the user experience, it can also be used to quantitatively compare the Reverse Dictionary to other similar products.

### 4.2 Running the RD-Accuracy Test

Identical tests were carried out on both the Reverse Dictionary and Onelook’s Reverse Dictionary (henceforth Onelook). The 80 tests compiled from volunteers’ input, as described previously, were used and the descriptor words entered manually into the dictionaries. Although scripts were written to automate the testing, they were not used as some manual changes were required to be made to the tests. Changes included spelling correction and the removal of unsuitable descriptor words (proper nouns not found in the dictionary for example). The location of the headword in the result set was recorded in a spreadsheet and formulae used to calculate the scores. In order to determine the location of the headword, the headword must have appeared on its own and not part of a compound word. For example, if the headword was “computer” and “personal computer” appeared at position 1 and “computer” at position 5; then 5 would have been accepted as the scored position.

To reduce the time taken to run the tests both user interfaces to the Reverse Dictionary were used together, simultaneously. Tests in which the headword did not appear or appeared after entry number 100 were awarded zero.
4.3 Results

- The Reverse Dictionary received an average score of 30.24
- Onelook received an average score of 48.95

The full test results can be found in Appendix B.

4.4 Discussion

The results show the Reverse Dictionary having a low score with Onelook having nearly 19% better score. This is not necessarily because of a poor implementation of the Reverse Dictionary and the reasons why will be explained here.

The RD-Accuracy Test scored the reverse dictionaries by where the headword occurred in a list of 100 results. Onelook always returns 100 results whereas the Reverse Dictionary returned only the most relevant results. Sometimes the headword was not returned at all because it was truncated from the result set. An analysis of the results shows that the Reverse Dictionary is less likely to score a hit (a score above 0), resulting in a lower average for the Reverse Dictionary.

Another reason for relatively poor performance of the Reverse Dictionary was due to the age of the data source. The LDOCE was published in 1978 before the advent of mobile telephones, the World Wide Web, wireless networking etc. Many entries in the test data set were related to computing (5 occurrences of the word “computer”), possibly as a result of computing students forming a large subset of the volunteers. Although there are computing related entries in the LDOCE, many of the descriptor words were either not found or not relating to computing, for example “wireless”, “chat” and “internet”. Even terms that were common in the late 70s are described by today’s English speaking population using more modern terms. Look for example at the entry for “telephone”; the descriptor words supplied by a volunteer were “digital, mobile, handset, call, buttons”. The LDOCE’s entry for telephone (noun) is “a method for sending sounds and talking to others over long distances by electrical means; the apparatus that receives or sends sound esp. speech”. Conversely Onelook indexes many data sources including Encarta, Wikipedia and Dictionary.com making it much more adept at understanding new and
or technical terms. This is not a deficiency in the Reverse Dictionaries’ Inference Engine but due to a lack of sufficient data sources. Similarly, the LDOCE’s definitions are composed only of the LDV’s 2851 words meaning that the links of the semantic network (built from the occurrence of words inside the definitions) were limited.

Another shortcoming of the RD-Accuracy Test is that it relies on the resulting headwords being sorted by similarity. The results from the Reverse Dictionary often have very similar similarity ratings. A typical result set will have one word with a rating of 4.5, a few with 3.5 and the rest with 3.0 or 2.0 (word with a lower similarity are removed from the result set). The words are ordered arbitrarily within their similarity measure groups. If the sought headword has a similarity rating of 3.0 it may be placed at position 4 or position 10 for example. The RD-Accuracy score will differ greatly, even though the similarity of the word does not change. This adds a variable in the gathering of test results which may have had a detrimental effect on the results. Because of the closed source nature of Onelook it is impossible to predict if a similar problem occurs for that reverse dictionary.

Another factor which affected the accuracy of the Reverse Dictionary is that errors in parsing the LDOCE introduced mistakes into the semantic network. It is, however hard to quantify this affect. Also, the User Interface splits the input string into separate words by looking for spaces (whitespace). Some headwords in the LDOCE contain more than one word, such as “personal computer”. By design, the User Interface does not allow searches including these compound words, limiting the accuracy of the Reverse Dictionary.

Although there are a number of issues surrounding the RD-Accuracy Test, it still provides a good baseline accuracy test. Given more time further work could be done on the test to make it more accurate but for the purposes of this project it is taken verbatim that Onelook performs better than the Reverse Dictionary.

### 4.4.1 Responsiveness

The responsiveness of a reverse dictionary is the time it takes to perform a search. No official benchmarking was done on the performance of the Reverse Dictionary, that is the time taken to produce a lookup but some observational results were noted. Using the User Interface, a lookup takes on average
10 seconds. Although not quick, this speed is acceptable to the user experience.

There are a few factors affecting the performance of the Reverse Dictionary. A relational database is not the most efficient method for storing and traversing a semantic network as there is considerable overhead. A custom data structure would have been much faster, but the other advantages of using the database, as mentioned previously, do outweigh this disadvantage. Also Python, being an interpreted language is slower to execute an application that a comparable compiled language such as C++. Onelook _may_ have been written in a compiled language, although this is purely speculative.
Chapter 5

Conclusions and Further Work

5.1 Conclusions

The overall performance of the Reverse Dictionary is slightly disappointing. This is not, however, viewed in a negative manner; as many reasons for this poor performance have been identified, future versions can be greatly improved. A working product has been delivered on time. Furthermore, the project management was carried out in a professional manner, technical difficulties were identified and overcome and many lessons were learnt.

5.2 Further Work

There is much room for improvement in this project. In this section some areas for improvement are highlighted and suggestions for future enhancements are made.

Firstly the semantic network could be expanded by adding additional data sources, particularly those
that are more up-to-date than the LDOCE. This would improve the accuracy of the Reverse Dictionary. Although WordNet was used in this project, it was not used as an independent data source, that is it was not added to the semantic network, and used only to test similarity between words.

Also the additional data sources could be drawn from bi-lingual dictionaries. Each headword would be looked up in the bi-lingual dictionary and each word in the definition then translated back into English, and its English equivalent added as a link in the semantic network.

One strong criticism made of the Reverse Dictionary was that all the links in the semantic network had equal weighting. This made it harder for the Inference Engine to sort result sets properly by their similarity rating. A corpus such as the Brown Corpus could be used to count word frequencies giving a higher weighting to more common words. Additionally word association can be measured by tracking co-occurrence in a given window size.

Another possible enhancement is to make the Reverse Dictionary adaptive. If the user were asked for feedback after each search to confirm the correct headword, this could be stored in the system. With this data it would be possible to give more weight to the path followed through the semantic network for similar searches. This will increase the responsiveness and accuracy of the Reverse Dictionary.

A novel idea for comparing the accuracy of the Reverse Dictionary is to use Google as another competitor. The RD-Accuracy test could be extended so that the test descriptor words are entered into Google and then search for the headword in the titles and descriptions returned.

The Inference Engine could be written in a compiled language such as C++ which would have to interface with the python modules. This would lower portability (as the program would have to be recompiled to run on a different platform) but would improve the responsiveness of the system.

Finally, an interesting extension could be a visualisation of the semantic network. A 3D graph is envisaged with the headwords as the nodes, rendered as spheres and the links shown as physical connections between the nodes. The size of the nodes could be scaled based on their occurrence frequency in a corpus and the size or colour of the links could change to show activity as a search is executed. The model could be navigated like a 3D virtual environment, perhaps using an immersive headset and joystick; or more simply with a keyboard, mouse and monitor. Apart from being a graphically pleasing project, the visualisation could provide a graphical insight into the use of the English language and the relationships...
between words.
I found the project both challenging and interesting. When I first had the idea for the project in February 2005 I imagined it as a tool that I could quickly hack together to help me when writing reports and emails but I did not have enough free time to commit to it. It was only in September of that year that I decided to take it on as my Final Year Project and started to explore the task ahead in detail. I did not imagine at the outset the complexity of the project or the difficulties which would arise when designing it.

With the guidance of my peers and teachers I set about the background reading for the project. This was particularly interesting as having not studied natural language processing before (AI32 is a second semester module) I was unfamiliar with the terminology used or with the difficulties of natural language processing as a whole. Of particular interest was a paper by George Demetriou[6] which uncovered some issues involved with using the LDOCE. Reading of this paper and many others also introduced me to the academic writing style, something I found initially very slow and hard to read but got used to over the course of the project.

With some ideas written down after the background reading and some more ideas gathered from dis-
cussions with my supervisor I designed and started to build the software. I was already familiar with the importance of a good design and the project experience reinforced that ideal. Building the software was an enjoyable experience as I find coding both fun and satisfying. As I encountered problems during coding and debugging I dealt with them in a professional manner, sometimes seeking help from those around me. I was impressed with the amount of help available from staff and students alike on the News Groups, by email and in person. I met with Katjar Markert in January who gave me some additional ideas for enhancements and testing.

The demonstration with the assessor was a valuable activity which gave me important feedback on the project and some more ideas for the project report. It also ensured that I had a working prototype ready before the end of the second term.

Particularly satisfying was using the Reverse Dictionary whilst writing this report to help me find words I was looking for. It reassured me that despite its shortcomings, I had managed to create a useful product.

Throughout the project I learnt the importance of making and keeping to a schedule and building redundancy into that schedule. Without it I would not have completed the project on time but I do feel that my schedule could have been more detailed to help in time management.

One thing I feel I failed to do was to dedicate any serious time to expansion beyond the minimum requirements. Although there was some expansion done, no time was set aside in the schedule and “spare” time had to be used to do so. I would recommend to future students that they allow a few weeks near the end of the project to expand upon their system. I would also recommend to future students that they ensure they set aside plenty of time over the Easter break to write the report, that is unless they enjoy working very long days.

I feel that information dissemination is very important in helping a final year project to proceed. I encountered a problem with decoding the LDOCE, which was supplied with no documentation. There was documentation available in the school on this matter but its presence was not known of and was only discovered by using google.

I feel that the project has put me in better stead to face the challenges of the workplace, with enhanced skills in time management, project management and, of course, software engineering.
In conclusion I have enjoyed the experience of creating and writing this project and am grateful to have had the opportunity to have done so.
Appendix B

Test Results

These are the full results of the RD-Accuracy test run on the Reverse Dictionary and Onelook.

Description of column headings:

**Number**  The test number  
**Word**  Headword  
**RD-Pos**  Position occurred in the Reverse Dictionary  
**RD-Score**  Score awarded to the Reverse Dictionary  
**OL-Pos**  Position occurred in Onelook  
**OL-Score**  Score awarded to Onelook

<table>
<thead>
<tr>
<th>Number</th>
<th>Word</th>
<th>RD-Pos</th>
<th>RD-Score</th>
<th>OL-Pos</th>
<th>OL-Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>phone</td>
<td>3</td>
<td>97</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Computer</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

38
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Football</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Beer</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>99</td>
</tr>
<tr>
<td>5</td>
<td>Generous</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>97</td>
</tr>
<tr>
<td>6</td>
<td>Gambling</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>Car</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>85</td>
</tr>
<tr>
<td>8</td>
<td>fun</td>
<td>0</td>
<td>0</td>
<td>33</td>
<td>67</td>
</tr>
<tr>
<td>9</td>
<td>Uniform</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>employee</td>
<td>13</td>
<td>87</td>
<td>3</td>
<td>97</td>
</tr>
<tr>
<td>11</td>
<td>cricket</td>
<td>1</td>
<td>99</td>
<td>4</td>
<td>96</td>
</tr>
<tr>
<td>12</td>
<td>cheese</td>
<td>2</td>
<td>98</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>game</td>
<td>152</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>bin</td>
<td>0</td>
<td>0</td>
<td>61</td>
<td>39</td>
</tr>
<tr>
<td>15</td>
<td>manage</td>
<td>14</td>
<td>86</td>
<td>2</td>
<td>98</td>
</tr>
<tr>
<td>16</td>
<td>abacus</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>97</td>
</tr>
<tr>
<td>17</td>
<td>nun</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>96</td>
</tr>
<tr>
<td>18</td>
<td>telephone</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>calendar</td>
<td>15</td>
<td>85</td>
<td>4</td>
<td>96</td>
</tr>
<tr>
<td>20</td>
<td>music</td>
<td>1</td>
<td>99</td>
<td>1</td>
<td>99</td>
</tr>
<tr>
<td>21</td>
<td>Job</td>
<td>36</td>
<td>64</td>
<td>1</td>
<td>99</td>
</tr>
<tr>
<td>22</td>
<td>teddy</td>
<td>7</td>
<td>93</td>
<td>2</td>
<td>98</td>
</tr>
<tr>
<td>23</td>
<td>book</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>89</td>
</tr>
<tr>
<td>24</td>
<td>conversation</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>97</td>
</tr>
<tr>
<td>25</td>
<td>Plant</td>
<td>2</td>
<td>98</td>
<td>5</td>
<td>95</td>
</tr>
<tr>
<td>26</td>
<td>football</td>
<td>66</td>
<td>34</td>
<td>2</td>
<td>98</td>
</tr>
<tr>
<td>27</td>
<td>robot</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>couch</td>
<td>2</td>
<td>98</td>
<td>10</td>
<td>90</td>
</tr>
<tr>
<td>29</td>
<td>chest</td>
<td>10</td>
<td>90</td>
<td>25</td>
<td>75</td>
</tr>
<tr>
<td>30</td>
<td>car</td>
<td>0</td>
<td>0</td>
<td>47</td>
<td>53</td>
</tr>
<tr>
<td>31</td>
<td>simple</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>house</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>99</td>
</tr>
<tr>
<td>33</td>
<td>monkey</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>34</td>
<td>composed</td>
<td>5</td>
<td>95</td>
<td>20</td>
<td>80</td>
</tr>
<tr>
<td>35</td>
<td>football</td>
<td>1</td>
<td>99</td>
<td>37</td>
<td>63</td>
</tr>
<tr>
<td>36</td>
<td>Malicious</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>37</td>
<td>cheese</td>
<td>0</td>
<td>0</td>
<td>25</td>
<td>75</td>
</tr>
<tr>
<td>38</td>
<td>Dog</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>helicopter</td>
<td>catnip</td>
<td>chocolate</td>
<td>ketchup</td>
<td>Container</td>
</tr>
<tr>
<td>---</td>
<td>------------</td>
<td>--------</td>
<td>-----------</td>
<td>---------</td>
<td>-----------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-----</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>------</td>
</tr>
<tr>
<td>75</td>
<td>smock</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>99</td>
</tr>
<tr>
<td>76</td>
<td>bank</td>
<td>6</td>
<td>94</td>
<td>1</td>
<td>99</td>
</tr>
<tr>
<td>77</td>
<td>camera</td>
<td>6</td>
<td>94</td>
<td>19</td>
<td>81</td>
</tr>
<tr>
<td>78</td>
<td>screwdriver</td>
<td>15</td>
<td>85</td>
<td>3</td>
<td>97</td>
</tr>
<tr>
<td>79</td>
<td>virus</td>
<td>0</td>
<td>0</td>
<td>27</td>
<td>73</td>
</tr>
<tr>
<td>80</td>
<td>snooker</td>
<td>35</td>
<td>65</td>
<td>3</td>
<td>97</td>
</tr>
<tr>
<td>TOTAL</td>
<td>30.24</td>
<td>48.59</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Appendix C

Test Data

This is the test data submitted by the volunteers for use in the RD-Accuracy test.

([phone],mobile,telephone,communication,speaking,text,cellphone,)
([Computer],Desktop,PC,Personal Computer,Laptop,Hand Held,Wireless,)
([Football],Leather,Player,Stitching,Spherical,Player,Stadium,)
([Beer],Glass,Drunk,Hops,Wheat,Alcohol,Golden,)
([Generous],Giving,Gratious.kind,charitable,helpful,willing,)
([Gambling],Chips,Money,Addiction,Cards,Odds,Stake,)
([Car],Passengers,Driver,Tyres,Road,Speed,Light,)
([fun],enjoyable,entertaining,exciting,)
([Uniform],Rank,Tradition,Smart,Discipline,Congruency,Rules,)
([employee],worker,subordinate,slave,company,employer,job,)
([cricket],insect,sport,ball,wicket,game,team,)
([cheese],milk,dairy,cow,stilton,mature,blue,)
([game],football,play,chess,ball,cheat,enjoy,)
([bin], trash, rubbish, waste, recycling, liner, refuse,)
([manage], run, administer, cope, supervise, direct,)
([abacus], chinese, counting, machine, beads, ancient, add,)
([nun], habit, catholic, celibate, female, monk, christ,)
([telephone], ring, call, communicate, voice, conversation, machine,)
([calendar], month, day, year, view, timetable, plan,)
([music], sound, melody, listen, compose, symphony, art,)
([Job], work, task, project, career, position, role,)
([teddy], brown, bear, child, toy, edward, fur,)
([book], paper, hardback, academic, reading, fictional, non-fictional,)
([conversation], discussion, speech, talk, discourse, socialisation, ideas,)
([Plant], leaves, roots, earth, pot, flower, grow,)
([football], kicking, soccer, game, hobby, sport, exercise,)
([robot], automated, electronic, computer, factory, explorer,)
([couch], sit, relax, sofa, lounge,)
([chest], box, jewellery, treasure, toy, cupboard, pirate,)
([car], motor, engine, steering wheel, brakes, accelerator, clutch,)
([simple], stupid, easy, unintelligent, bare, simpleton, childlike,)
([house], dwelling, building, home, abode, bungalow, flat,)
([monkey], animal, gorilla, ape, king kong, hairy,)
([composed], calm, collected, wrote, gathered, music, orchestra,)
([football], player, pitch, stadium, referee, soccer, team,)
([Malicious], Cruel, Violent, Sadistic, Sociopathic,)
([cheese], milk, butter, cheddar, stilton, dairy, churn,)
([Dog], Animal, Pet, Hound, Legs, Tail, Bark,)
([helicopter], fly, aircraft, travel, machine, rotor, pilot,)
([catnip], cat, plant, mint, rolling, feline,)
([chocolate], sweet, confectionery, candy, flavour, colour, flavour,)
([ketchup], red, tomato, sauce, condiment,)
([Container], plastic, contents,)
([telephone], digital, mobile, handset, call, buttons,)
([bank], sloped, ground, river, edge, side,)
([camera], picture, photograph, take, film,)
([screwdriver], tool, screw, handle, cross, flat,)
([virus], organism, pathogen, illness, flu,)
([snooker], sport, game, balls, cue, table,)
Bibliography


