THE AUSTRALIAN NATIONAL UNIVERSITY

TR-CS-02-05

Febrl - Freely extensible biomedical record linkage.

Peter Christen and Tim Churches

October 2002

Joint Computer Science Technical Report Series

Department of Computer Science
Faculty of Engineering and Information Technology

Computer Sciences Laboratory
Research School of Information Sciences and Engineering

This technical report series is published jointly by the Department of Computer Science, Faculty of Engineering and Information Technology, and the Computer Sciences Laboratory, Research School of Information Sciences and Engineering, The Australian National University.

Please direct correspondence regarding this series to:

Technical Reports
Department of Computer Science
Faculty of Engineering and Information Technology
The Australian National University
Canberra ACT 0200
Australia

or send email to:

`Technical.Reports@cs.anu.edu.au`

A list of technical reports, including some abstracts and copies of some full reports may be found at:

`http://cs.anu.edu.au/techreports/`

Recent reports in this series:

TR-CS-02-04 John N. Zigman and Ramesh Sankaranarayana. *djvm - a distributed jvm on a cluster*. September 2002.

TR-CS-02-03 Adam Czezowski and Peter Christen. *How fast is -fast? performance analysis of kdd applications using hardware performance counters on ultrasparc-iii*. September 2002.

TR-CS-02-02 Adam Czezowski Bill Clarke and Peter Strazdins. *Implementation aspects of a sparc v9 complete machine simulator*. February 2002.

TR-CS-02-01 Peter Christen and Adam Czezowski. *Performance analysis of kdd applications using hardware event counters*. February 2002.

TR-CS-01-02 Jeremy E. Dawson and Rajeev Gore. *Mechanising cut-elimination for display logic*. October 2001.

TR-CS-01-01 Stephen Roberts Peter Christen, Markus Hegland and Irfan Altas. *A scalable parallel fem surface fitting algorithm for data mining*. October 2001.

Febrl – Freely extensible biomedical record linkage

Release 0.1.1

Peter Christen [1], Tim Churches [2]

October 16, 2002

[1] Department of Computer Science
Australian National University
Canberra ACT 0200
Australia
Email: peter.christen@anu.edu.au

[2] Centre for Epidemiology and Research
New South Wales Department of Health
Locked Mail Bag 961
North Sydney NSW 2059 Australia
Email: tchur@doh.health.nsw.gov.au

Abstract

This manual describes prototype software called **Febrl** designed to undertake probabilistic data cleaning (or standardisation) and record linkage. Written in the **Python** programming language, this software aims to allow health, biomedical and other researchers to clean (standardise) and link data sets of all sizes faster, with less effort and with improved quality.

The authors would be grateful if users of **Febrl** would inform us (by e-mail) of how they have used the system. We are particularly interested in references to scientific papers or reports which mention or cite **Febrl**. Some suggested citations for **Febrl** will be provided in the next version of this manual.

See Also:

- **Febrl** *Project Web Site*
  (http://datamining.anu.edu.au/linkage.html)
  for information on this project
- *Python Web Site*
  (http://www.python.org/)
  for information on Python

CONTENTS

CHAPTER

ONE

Acknowledgments

This project is equally funded by the *Australian National University (ANU)* and the *NSW Department of Health* under an *AICS (ANU-Industry Collaboration Scheme) AICS #1-2001.* The authors would like to thank everybody who supported this project and helped to make it happen: Markus Hegland, Ole Nielsen, Stephen Roberts and David Bulbeck. We are specially grateful to Justin Zhu (Department of Computer Science, ANU) for his recognition of the potential of hidden Markov models in data standardisation, and to Kim Lim (Centre for Epidemiology and Research, NSW Department of Health) for her input into the design of the software and in helping to test and debug it.

CHAPTER

TWO

Introduction

Record linkage is a rapidly growing field with applications in many areas of health and biomedical research [1, 6, 9]. It is an initial step in many epidemiological studies and data mining projects, in order to assemble the required data in a form suitable for analysis. Data mining aims to analyse large and complex data sets to find patterns and rules, to detect outliers or to build predictive models of such data sets [7]. Often the data required for such analyses are contained in two or more separate databases, which do not share a common unique record identifier (key). In such cases, record linkage techniques need to be used to join the data.

Methods used to tackle the record linkage problem fall into two broad categories. *Deterministic* methods in which sets of often very complex rules are used to classify pairs of records as *links* (i.e. relating to the same person or entity) or as *non-links*; and *probabilistic* methods in which statistical models are used to classify record pairs. Probabilistic methods can be further subdivided into those based on *classical* probabilistic record linkage theory as developed by *Fellegi and Sunter* [4] in 1969, and newer approaches using maximum entropy and other machine learning techniques.

Historical collections of administrative and other health data nowadays contain many tens or even hundreds of millions of records, with new data being added at the rate of millions of records per annum. Although computing power has increased tremendously in the last few decades, large-scale record linkage is still a slow and resource-intensive process. There have been relatively few advances over the last decade in the way in which probabilistic record linkage is undertaken, particularly with respect to the tedious *clerical review* process which is still needed to make decisions about pairs of records whose linkage status is doubtful. Unlike computers, there has been no increase in the rate at which humans can undertake these clerical tasks.

Warning: Probabilistic record linkage is a powerful technique which can be used to assemble data sets which would not otherwise be available for health and biomedical research. However, there is the potential for the invasion of personal privacy whenever linkage between data sets is undertaken. It is therefore imperative that record linkage is performed in a strictly controlled and secure environment under one or more of the following conditions:

- where informed consent has been given for the linkage to take place by all the individuals whose personal data is to be linked;
- where a properly constituted institutional ethics committee has given permission for the linkage to take place because it considers that the public good which will result from the research substantially outweighs the public interest in the protection of privacy;
- where legislation specifically permits or mandates the linkage of particular data files.

Users of the **Febrl** system should take time to familiarise themselves with all legislation, regulations, guidelines and procedures which relate to the protection of privacy and confidentiality, or which otherwise govern linkage between data collections in their jurisdiction. References to relevant Australian legislation and guidelines can be found on the **Febrl** project Web site at http://datamining.anu.edu.au/linkage.html.

The programs described in this manual, known collectively under the moniker *'Freely extensible biomedical record linkage'* (**Febrl**), are currently being developed as part of a collaborative project being undertaken by the *ANU Data Mining Group* and the *Centre for Epidemiology and Research* in the *New South Wales Department of Health*. The aim of the project is to develop improved techniques for probabilistic record linkage which combine *classical* probabilistic methods with deterministic and, in particular, machine learning techniques in order to improve the linkage quality and to reduce the incredibly time consuming and tedious manual clerical review process of possible links. Additionally, the project intends to make good use of modern high-performance parallel computing platforms, such as clusters of commodity PCs or workstations (which can be used as virtual parallel computers with some additional software installed), multiprocessor servers or supercomputers. We hope that the resulting software will allow biomedical and other researchers to link data sets of all sizes more efficiently and at reduced costs.

The **Febrl** program code and associated documentation and data files are published under the *ANU Open Source License* (see Appendix G), which is derived from the *Mozilla Public License version 1.1* with minor changes to make it suitable for Australian law. The license permits the free use and redistribution of the **Febrl** manual (the document you are now reading) and free use, modification and re-distribution of the associated **Febrl** programs and data files, provided that any modifications or enhancements to the program code are made freely available to other users of the programs under the same licensing arrangements. You are strongly urged to read the license before you start using the programs. Please pay particular attention to the *DISCLAIMER OF WARRANTY* which appears in the license.

We hope that release of the programs under an open source license will encourage other researchers to contribute to the ongoing development of the system, and to share the responsibility for its maintenance and support. At this stage, there are many areas of the system which need further work – some of these are listed in Appendix D.

For this initial version, it is assumed that the reader has at least superficial familiarity with the syntax of the Python programming language in order to customise the project configuration module `project.py`, which, like the rest of the system, is written in Python. Later versions of the system may provide configuration tools which remove this requirement. Of course, knowledge of Python will be necessary if you wish to extend or customise the system. However, as well as being very powerful, Python is also extremely easy to learn, even for people with little or no prior programming experience. Python tutorials as well as implementations of the language itself can be found on the Python Web site at http://www.python.org web site. Python is a free, open source language which can be downloaded, installed and used on any number of computers for any purpose without charge. Versions of Python are available for all popular operating systems and types of computer.

The structure of this manual is as follows. The next Chapter gives a short overview of the techniques and applications of record linkage and data cleaning and standardisation in general. An overview of the **Febrl** system is given in Chapter 4, followed by a brief introduction to hidden Markov models (HMMs) in Chapter 5. HMMs are used in the **Febrl** data cleaning and standardisation module **pyStandard.py** (short for *Pythonic Standardiser*). They are a powerful alternative to the often cumbersome rules-based approach to data standardisation. A detailed description of the data cleaning and standardisation process and instructions on the usage of the **pyStandard.py** program is given in Chapter 6. Chapter 7 deals with the issue of HMM training. Instructions for the use of the programs **pyTagData.py** and **pyTrainHMM.py** are given in this chapter. A brief description of various components of the record linkage process is given in Chapter 8. Much fuller descriptions of the program modules used for record linkage will be added to Chapter 8 in subsequent versions of this manual. Currently one auxiliary program (**pyRandomSelect.py** which allows random selection of input records) is provided and described in Chapter 9. Configuring the **Febrl** system using the configuration module `project.py` is explained in Chapter 10, and the file format of various look-up and frequency table files used by the system is the topic of Chapter 11. The installation of the **Febrl** system is discussed in Chapter 12. In Appendix A lists of all defined hidden Markov model states are given and Appendix B contains the list of all supported tags used in the data standardisation process. The manifest in Appendix C gives a list of all files contained in the current version of the **Febrl** distribution. A list of outstanding development tasks and planned additions and enhancements to the system appears in Appendix D. Finally, a copy of the ANU Open Source License can be found in Appendix G.

Note: The authors recognise that some aspects of the **Febrl** project may have application in certain business and commercial settings. Such use is permitted by the ANU Open Source License under which **Febrl** is licensed. However, we wish to emphasise that the software is being developed purely with the needs of health and biomedical researchers in mind, and there are no plans to add features which business users might specifically need, such as *Australia Post Address Matching Approval System (AMAS)* processing and certification. See http://www.auspost.com.au/BCP/0,1080,CH2403%257EMO19,00.html for more information on AMAS and related technologies. It should also be remembered that the **Febrl** project is still in the early stages of its development, and the software cannot be considered to be of production quality.

We urge users with business or commercial data processing needs to examine the wide range of products and services available from commercial vendors. A non-comprehensive set of links to the Web sites of vendors of business-oriented data quality and data cleaning software services is available on the **Febrl** project Web site at http://datamining.anu.edu.au/linkage.html. The links are provided for information only and do not imply endorsement or recommendation of any particular vendor's products or services. Vendors of relevant products or services who would like a link to their Web site to be added to the **Febrl** project Web site should contact the authors by email.

CHAPTER

THREE

Record Linkage and Data Cleaning

Record linkage techniques are used to link data records relating to the same entities, such as patients or customers. Record linkage can be used to improve data quality and integrity, to allow re-use of existing data sources for new studies, and to reduce costs and effort in data acquisition for research studies.

If a unique identifier or key is available in all of the data sets to be linked, then the problem of linking at the entity level is trivial - a simple *join* operation in SQL or its equivalent in other data management systems is all that is required. However, if no unique key is shared by all of the data sets, then various record linkage techniques need to be used. As discussed in the previous chapter, these techniques can be broadly classified into *deterministic* or rules-based approaches, and *probabilistic* approaches.

No matter what technique is used, a number of issues need to be addressed when linking data. Often, data is recorded or captured in various formats, and data items may be missing or contain errors. A pre-processing phase that aims to clean and standardise the data is therefore an essential first step in every linkage process. Data sets may also contain duplicate entries, in which case linkage may need to be applied within a data set to de-duplicate it before linkage with other files is attempted.

The process of linking records has various names in different user communities. While epidemiologists and statisticians speak of *record linkage*, the same process is often referred to as *data matching* or as the *object identity problem* [8] by computer scientists, whereas it is sometimes called *merge/purge processing* or *list washing* in commercial processing of customer databases or mailing lists. Historically, the statistical and the computer science communities have developed their own techniques, and until recently few cross-references could be found. In this Chapter we give an overview and try to identify similarities in the extant methods.

Computer-assisted record linkage goes back as far as the 1950s. At this time, most linkage projects were based on *ad hoc* heuristic methods. The basic ideas of probabilistic record linkage were introduced by *Newcombe and Kennedy* [13] in 1962 while the theoretical foundation was provided by *Fellegi and Sunter* [4] in 1969. Using frequency counts [21] to derive agreement and disagreement probabilities, each pair of fields of each pair of records is assigned a match weight, and critical values of the sum of these match weights are used to designate a pair of records as either a *link*, a *possible link* or a *non-link*. Possible links are those pairs for which human oversight, also known as *clerical review*, is needed to decide their final linkage status. In theory, the person undertaking this clerical review has access to additional data (or may be able to seek it out) which enables them to resolve the linkage status. In practice, often no additional data is available and the clerical review process becomes one of applying *human intuition* or *common sense* to the decision based on available data. One of the aims of the **Febrl** project is to automate (and possibly improve upon) this process through the use of machine learning and data mining techniques.

To reduce the number of comparisons (potentially each record in one data set has to be compared with every record in a second data set), *blocking techniques* are typically used. The data sets are split into smaller blocks using blocking variables, like the postcode or the *Soundex* encoding of surnames. Only records within the same blocks are then compared. To deal with typographical variations and data entry errors, approximate string comparison functions [15] are often used for names and addresses. These comparators usually return a score between 0.0 (two strings are completely different) and 1.0 (two strings are the same).

In recent years, researchers have been exploring the use of machine learning and data mining techniques [19] both to improve the linkage process and to allow linkage of larger data sets. For very large data sets, with hundreds of millions of records, special techniques have to be applied [20] to be able to handle such large volumes of data. Sorting large number of records becomes the main bottleneck, so extracting subsets of possible links from an unsorted large data file [22] has to be done as a pre-processing step before the actual record comparisons can be done.

The terms *data cleaning* (or *data cleansing*), *data standardisation*, *data scrubbing*, *data pre-processing* and *ETL* (extraction, transformation and loading) are used synonymously to refer to the general tasks of transforming the source data (often derived from operational, transactional information systems) into clean and consistent sets of records which are suitable for record linkage or for loading into a data warehouse [17]. The meaning of the term *standardisation* in this context is quite different from its use in epidemiology and statistics, where it usually refers to a method of dealing with the confounding effects of age. The main task of data standardisation in record linkage is the resolution of inconsistencies in the way information is represented or encoded in the data. Inconsistencies can arise through typographical or other data capture errors, the use of different code sets or abbreviations, and differences in record layouts.

Fuzzy techniques and methods from information retrieval have been used to address the record linkage problem, with varying degrees of success. One approach is to represent text (or records) as document vectors and compute the *cosine distance* [3] between such vectors. Another possibility is to use an *SQL* like language [5] that allows approximate joins and cluster building of similar records, as well as decision functions that decide if two records represent the same entity. Other methods [11] include statistical outlier identification, pattern matching, clustering and association rules based approaches. Sorting data sets (to group similar records together) and comparing records within a sliding window [8] is a technique similar to blocking as applied by traditional record linkage approaches. The accuracy of the matching can be improved by having smaller window sizes and performing several passes over the data using different (often compound) keys, rather than having a large window size but only one pass. This corresponds to applying several blocking strategies in a record linkage process.

Even though most approaches described in the computer science literature use approximate string comparison operators and external look-up tables to improve the matching quality, none considers the statistical theory of record linkage as developed by *Fellegi and Sunter* [4] and improved and extended by others. The **Febrl** system uses this approach as the basis of its record linkage engine, although deterministic and machine learning techniques may also be added at a later date.

Of course, the problem of finding similar entities not only applies to records which refer to persons. Increasingly important is the removal of duplicates in the results returned by Web search engines and automatic text indexing systems, where copies of documents have to be identified and filtered out before being presented to the user.

CHAPTER

FOUR

System Overview

Record linkage consists of two main steps. The first one deals with data cleaning and standardisation, while the second performs the actual linkage. The current version of the **Febrl** system only contains programs for data cleaning and standardisation pre-processing A record linkage module will be added later.

The aim of the data cleaning and standardisation process is to transform the information stored in the original data into a well defined and consistent form. Personal information may be recorded or captured in various formats, spelled differently, it might be outdated, some items may be missing or contain errors. For example, if data is captured over the telephone, spelling variations of names are common. Typing errors happen frequently when dates are entered. The data cleaning steps attempt to deal with these problems. Conversion of the original input data into a well defined form, and segmenting it into many smaller *output fields*, allows the linkage process to be much more accurate.

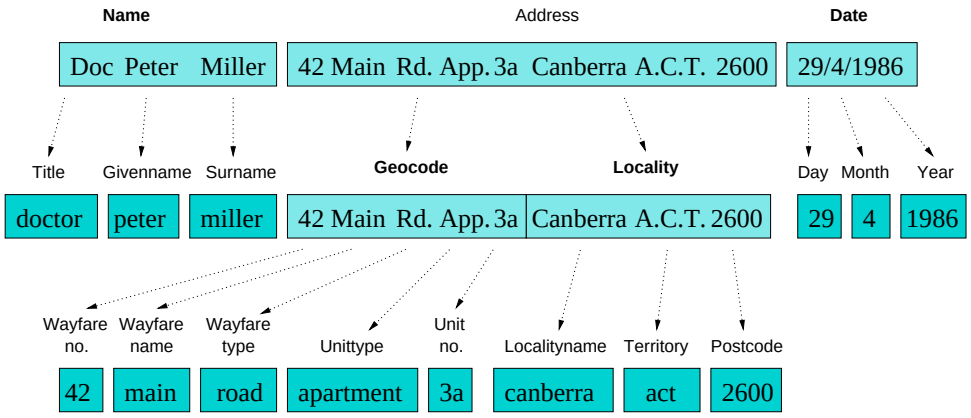

Figure 4.1: Example of a personal information standardisation.

As an example, the record in Figure 4.1 is cleaned and split into 14 output fields. Comparing these output fields individually with output fields of other records results in a much better linkage quality than just comparing for example the whole name or the whole address as a string with the name from another record.

The data cleaning and standardisation is implemented in the **pyStandard.py** program. A number of look-up table and correction list files are used for standardisation processing (these are described in detail in Chapter 6). However, **Febrl** data cleaning and standardisation primarily employs a supervised machine learning approach implemented through a novel application of *hidden Markov models* (HMMs). A brief introduction to HMMs and their use for data standardisation is given in the following Chapter. Before data standardisation can be performed with a given data set, the user needs to *train* HMMs using *training data* from the same or similar data sets. Two HMMs need to be trained, one for names and one for addresses (geocode[1] and locality). The process of creating training data is described in

[1] The term *geocode* is used as a noun in this manual to refer to the *street address* part of an address - typically a *street address* comprises a wayfare number, a wayfare name and a wayfare type as shown in the diagram on this page. When used as a verb, *to geocode* refers to the process of assigning a latitude and longitude and/or some geographical area identifier or

Chapter 7. Once HMMs are available for a given data set (or class of data sets), the data cleaning and standardisation process becomes easy and efficient.

Note: The linkage module is currently under development, and details will be added as soon as it becomes available.

other geographical attribute to a particular address.

CHAPTER

FIVE

Hidden Markov Models for Data Standardisation

Traditional data cleaning and standardisation programs have used various rule-based approaches to the task of parsing raw data. Typically, programmers have used *regular expressions* (as implemented in tools such as **awk**, **grep** or **agrep**), or other pattern-matching languages such as **SNOBOL** to search for particular *signatures* in the input data in order to work out how to segment it. However, pattern-matching languages in general and regular expressions in particular are not for the faint-hearted.

> *Some people, when confronted with a problem, think "I know, I'll use regular expressions". Now they have two problems.* – Jamie Zawinski, in `comp.lang.emacs`

The **AutoStan**[1] [12] program improved on simple (or far-from-simple) regular expressions by using an initial lexicon-based tokenisation phase followed by a re-entrant rule-based parsing and data re-writing phase. The **Febrl** system also uses lexicon-based tokenisation, but then uses a probabilistic approach based on *hidden Markov models* (HMMs) [16] to assign each word in the input string to a particular output field.

HMMs were developed in the 1960s and 1970s and are widely used in speech and natural language processing [16]. The use of HMMs in data standardisation was the topic of two recent research papers. Borkar et al. [2] present a nested HMM approach for *text segmentation* (which is the task of segmenting an input string into well defined output fields, so basically the same as data standardisation) of Asian and American addresses and bibliographic records, and Seymore et al. [18] discuss how the structure of HMMs can be learned from example data for the task of *information extraction* (e.g. extracting names, titles and keywords from publication abstracts).

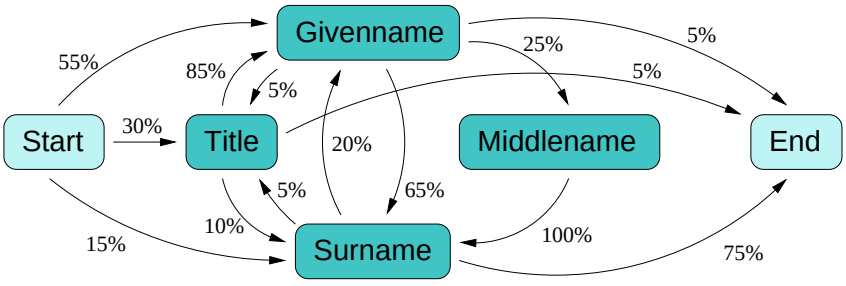

Figure 5.1: Example hidden Markov model for names.

An HMM is a probabilistic finite state machine made of a set of unobserved (hidden) states, transition edges between these states and a finite dictionary of discrete observation (output) symbols. Each edge is associated with a transition probability, and each state emits observation symbols from the dictionary with a certain probability distribution. Transition and observation probabilities are stored in two matri-

[1] **AutoStan** and **AutoMatch** as formerly sold by *MatchWare Technologies* are now part of the *Ascential Integrity (R)* product line. See http://www.ascentialsoftware.com

ces, as shown in the two tables below. The sum of all transition probabilities out of a given state has to equal 1.0, as does the sum of all observation probabilities for a particular state.

Figure 5.1 shows a simple HMM example for names with six states, including the *start* and *end* states (which are both virtual states that are not actually stored in an HMM as no observation symbols are emitted in these states). A list of initial state probabilities is used instead of the *start* state (i.e. probabilities that give the likelihood of a sequence starting in a certain state). In the given example HMM, there is a probability of 0.55 that a name starts with a *Givenname* and is followed with a (conditional) probability of 0.65 by a *Surname*, or a probability of 0.25 by a *Middlename*, and so on.

Table 5.1: Example name HMM transition probabilities.

	To state					
From state	*Start*	Title	Givenname	Middlename	Surname	*End*
Start	–	0.30	0.55	0.0	0.15	–
Title	–	0.0	0.85	0.0	0.10	0.05
Givenname	–	0.05	0.0	0.25	0.65	0.05
Middlename	–	0.0	0.0	0.0	1.00	0.0
Surname	–	0.05	0.20	0.0	0.0	0.75
End	–	–	–	–	–	–

Table 5.2: Example name HMM observation probabilities.

	State					
Observation	*Start*	Title	Givenname	Middlename	Surname	*End*
TI	–	0.96	0.01	0.01	0.01	–
GM	–	0.01	0.35	0.33	0.15	–
GF	–	0.01	0.35	0.27	0.14	–
SN	–	0.01	0.09	0.14	0.45	–
UN	–	0.01	0.20	0.25	0.25	–

In the **Febrl** data standardisation system, instead of using the original words, numbers and other elements from the input records directly as observation symbols, each element (token) in the input record is *tagged* (as discussed in Chapter 6) using various look-up tables and rules, and these tags are used as the HMM observation symbols. This is done to make the derived HMMs more general, i.e. to allow HMMs to be trained on one data set and then be used with other similar, but distinct data sets, with little or no loss of performance, while still taking advantage of readily-available information such as lists of suburb and town (locality) names and postal codes. This differs from the approach taken by Borkar et al. [2] in which words in the input data are classified only by their type (numeric, alphanumeric or purely alphabetic), not their value, before an HMM is fitted to the data. In **Febrl** the input data is automatically tagged first using look-up tables (lexicons) and some simple rules. The result is a sequence of tags (one tag per input word or number), and these tag sequences are given to the HMM, which assigns them to its states (one tag per state). For a given tag sequence the most probable path through the HMM is determined using the *Viterbi* algorithm [16].

Let's take an example to make this a bit clearer. Assume an input record contains the name component

```
'doctor peter paul miller'
```

In a first step this input string is cleaned and then tagged (as described in Chapter 6). Assume the possible tags for names are `'TI'` (title words), `'GF'` (female given names), `'GM'` (male given names), `'SN'` (surnames) and `'UN'` for unknown words, as used in the name HMM example above. If the word `'doctor'` is found in the title look-up table, it is assigned a `'TI'` tag. Assuming the name `'peter'` is found in both the male given name and the surname look-up tables, it is assigned the two tags `'GM'` and `'SN'`. The tag `'GM'` is given to the name `'paul'` assuming it is only found in the male given name look-up table, and `'miller'` is assigned a `'SN'` tag assuming it is found in the surnames look-up table.

Because 'peter' was assigned two tags, the possible permutations of the input tags are the two tag sequences

```
['TI', 'GM', 'GM', 'SN']
['TI', 'SN', 'GM', 'SN']
```

These two tag sequences are given to the example name HMM and the Viterbi algorithm computes the probability for the most likely path through the model for each sequence. For example, the tag sequence ['TI', 'GM', 'GM', 'SN'] can be assigned to the following path through the HMM (with the corresponding observation symbols in brackets)

```
Start -> Title (TI) -> Givenname (GM) -> Middlename (GM) -> Surname (SN) -> End
```

The resulting probability of this path is

```
0.30 * 0.96 * 0.85 * 0.35 * 0.25 * 0.33 * 1.00 * 0.45 * 0.75 = 0.0023856525
```

with 0.30 being the transition probability from state Start to state Title, then 0.96 being the probability that the symbol 'TI' is observed in state Title and so on. Another possible path through the HMM for the same tag sequence would be

```
Start -> Title (TI) -> Surname (GM) -> Givenname (GM) -> Surname (SN) -> End
```

which would result in a probability of

```
0.30 * 0.96 * 0.10 * 0.15 * 0.20 * 0.35 * 0.65 * 0.45 * 0.75 = 0.0000663390
```

So for each tag sequence the Viterbi algorithm returns the most likely path with the corresponding probability, and the tag sequence with the highest probability and the corresponding path through the HMM is taken. The input words are then associated with the corresponding output states, in this example 'doctor' will become the title, 'peter' will become the given name, 'paul' the middle name and 'miller' the surname.

Both the transition and observation probabilities have to be *trained* using *training data* sets, i.e. collections of records which are taken from the same (or a similar) data set which will be used for data standardisation and that have been annotated manually. A HMM thus *learns* the characteristics of a data set. In initial experiments we found that around 1,000 training records taken randomly from a one million record data set (i.e. a 0.1% sample) results in a standardisation accuracy of more than 95%. Further improvement in the accuracy is possible as more examples are added to the training data sets.

Thus, instead of requiring highly trained programming staff to maintain a large number of complex rules written using arcane regular expression syntaxes to handle the myriad of special cases and exceptions which occur in real-life data, the data needed to train the HMMs used by **Febrl** can easily be created by clerical staff within a couple of days. Furthermore, this training process can be accelerated by *bootstrapping* it with training data sets derived from other, similar data sources. The process of HMM training for the **Febrl** system is explained in more detail in Chapter 7.

See Also:

- An introductory tutorial on hidden Markov models is available from the
  University of Leeds (UK)
  (http://www.scs.leeds.ac.uk/scs-only/teaching-materials/HiddenMarkovModels/html_dev/main.html)
- Slides of another introductory presentation on hidden Markov models by
  Michael Cohen, Boston University
  (http://screwdriver.bu.edu/cn760-lectures/l9n/l9_files/v3_document.htm)
- The hidden Markov model module simplehmm.py provided with the **Febrl** system is a modified re-implementation of *LogiLab's* Python HMM module. Please see
  Logilab HMM web site
  (http://www.logilab.org/hmm/)

CHAPTER

SIX

Data Cleaning and Standardisation using 'pyStandard.py'

Personal attributes (data items) used for record linkage can be broadly categorised into five classes: *names*, *addresses*, *dates* (such as date of birth) and *times*, *categorical attributes* (such as sex or country of birth) and scalar quantities (such as height or weight). The primary criteria for such attributes is that they are relatively invariant over time – they should not change, or at least not change often. For these reasons attributes such as diagnoses or procedures, or textual narratives of medical findings, are generally not used for record linkage purposes. Similarly, scalar attributes are also rarely used because they are subject to change, although it depends on the specific application. Currently **Febrl** provides specific facilities for the processing of names, addresses and dates. Later versions will provide facilities for the transformation of coded and uncoded categorical attributes into standard formats and values. In the meantime, the Python programming language in which **Febrl** is implemented can be used to write special purpose data transformation and cleaning functions or routines. By adopting a more object-oriented approach, the next version of **Febrl** will make it even easier to integrate custom data transformation procedures written by end users with other aspects of **Febrl** processing. Please see Appendix D for more details of these planned developments.

Figure 4.1 shows an example of a record containing personal information. As can be seen, each component is further split up into smaller *output fields* (see Table 6.1) each containing a basic piece of information, like a title, given name or postcode. These output fields may then be subject to further transformation in order to further standardise the values they contain before using them in the record linkage step.

The data cleaning and standardisation process in **Febrl** consists of the following three steps:

1. **Data Cleaning**
   First, all letters are converted into lower case, then various general corrections are performed using correction lists. For example, variations of *known as*, such as `'a.k.a.'` or `'aka'` are all replaced with a *standard* string `'known as'`.

2. **Data Tagging**
   The cleaned input string is split at space boundaries into a list of words, numbers and separators. Using various look-up tables, each element of this list is then assigned one or more *tags* (a list of all possible tags can be found in Appendix B). At the same time, each list element (which may itself be a tuple of words) is replaced by a corrected version from the look-up table if such a corrected version exists. A title word like `'doctor'` for example is assigned a title tag `'TI'` and it will be replaced with the word `'dr'`, as are the words `'md'` and `'phd'`. The look-up tables are searched using a *greedy* matching algorithm, which searches for the longest tuple of elements which match an entry in the look-up tables. For example, the tuple of words `('macquarie','fields')` will be matched with an entry in a look-up table for the locality `'macquarie fields'`, rather than with the shorter entry `'macquarie'` from the same look-up table.

3. **Data Segmenting**
   The list of tags is used to segment the input word elements into the correct *output fields*. It is intended to allow the use of either rules-based or probabilistic approaches (i.e. *hidden Markov models* as described in the previous Chapter) to assign tagged words to *output fields*, or a combination of both in a hybrid approach. Currently, both rule-based and HMM-based processing is supported

for names, but only HHM-based processing is available for addresses (which tend to have more complex and variable layouts and formats than do names). However, results of the HMM-based address processing are sufficiently good that it is unlikely that rule-based address processing will be implemented (at least by the authors).

The data cleaning and standardisation program **pyStandard.py** contains routines for all three steps, and it also deals with opening and loading the input file, pre-processing the input records and writing the clean and standardised records into an output file.

In the current version of **Febrl**, It is assumed that the input data is available as a text file and contains one record per line. Various input formats are supported:

- Comma-separated values (CSV), with and without quotes.
- Tab-separated values, with and without quotes.
- Column-formatted data, i.e. each record is a vector of bytes (characters) with each field defined by an offset from the beginning of the vector and a length.

The same formats are also supported for the output file. Input and output file types can differ, so it is possible to read data from a quoted tab-delimited file and write into a non-quoted comma-separated file. The only current limitation is that each text line corresponds to one record, and that all records must have the same format (number of fields or column ranges). Multi-line records and hierarchical database formats are not currently supported. It is possible to skip over header lines at the beginning of the input file. The input and output file types can be specified in the configuration module `project.py` (see Chapter 10).

Note: We plan to add support for *SQL* and *ODBC* database access (both for input and output) in a later version of this software.

Note: Parallelisation of the data cleaning and standardisation process is currently under development, for symmetrical multiprocessing (SMP) machines, dedicated clusters and ad-hoc clusters of workstations (COWs) as well as other types of supercomputer. Detailed description on how to run the data cleaning and standardisation in parallel will be added when these features become available.

As already noted, each input record is assumed to correspond to one line in the input file. Such a record is normally structured as several fields, e.g. one for the name, one for the address (or often one for street-address and one each for postcode, locality and state), and maybe one or two for dates of birth or other demographic data (such as country of birth). In some cases the input can be a free format string, while in other data sets it is already highly structured as individual components. The first pre-processing step in the **pyStandard.py** program consists of extracting and concatenating fields and sub-strings that correspond to the components *name*, *geocode*, *locality* and *date*. Currently up to two date components are supported. Next, these components are handed off to the corresponding standardisation routines that segments words and numbers into the appropriate *output fields*. Finally, these output fields are combined into an output record and written to the output file according to the output specifications given in the `project.py` module. Additionally, log and error information can be saved into a log file and printed out in verbose mode. In Section 6.6 the command line arguments and options for the **pyStandard.py** program are presented in more detail.

Note: It can be argued, cogently, that in circumstances where the street address is already segmented into components, such as wayfare (street) number, wayfare name, locality (suburb or town) and postal code, it does not make much sense to concatenate these components and then try to parse back into individual components again. Future versions of **pyStandard.py** will add support for standardising such already-segmented data. However, in real life, things are often not so clear cut, and often data items are entered in the wrong fields or spill over from one field to the next. In these circumstances, it may be advantageous to re-combine all the address and/or names elements and re-segment them using **pyStandard.py**.

Table 6.1: Supported output fields.

Name	Geocode	Locality	Date
title	wayfare_number	postcode	day1
gender_guess	wayfare_name	locality_name	month1
givenname	wayfare_type	locality_qualifier	year1
alt_givenname	wayfare_qualifier	territory	day2
surname	unit_number	country	month2
alt_surname	unit_type		year2
	property_name		
	institution_name		
	institution_type		
	postaddress_number		
	postaddress_type		

6.1 Output Fields

An input record is standardised and its resulting cleaned elements are stored in a table of *output fields*, as shown above. Note that these output fields can contain more than one word, or they can be empty if the input record did not contain a corresponding input element.

6.2 Step 1: Data Cleaning

The input to the data cleaning routine is a string that contains an input component, i.e. either a *name* or an *address* (*date* components are handled differently, see Section 6.5). First, all letters in such a string are converted into lower case. Secondly, a list of replacement strings is used to replace certain words, abbreviations and characters with others. This list is loaded from a correction list file (see Section 11.1 for the details of the formats of these files). Each entry in such a list is made of a string (that can be one or more words,or a simple character) and a corresponding replacement string. For each entry in the list, the input string is scanned and if an original string is found it is replaced by the corresponding replacement string.

Table 6.2: Example correction list.

Original	Replacement
' knownas '	' known as '
' a.k.a. '	' known as '
' aka '	' known as '
' babyof '	' baby of '
' b/o '	' baby of '
' b.o. '	' baby of '
' n/a '	' '
' na '	' '
'['	'__'
'('	'__'
':'	' '

Each correction list is sorted and processed by decreasing length of the original (target) string, i.e. long target strings are searched for and replaced first. In the example correction list below, the entry `' knownas '` would be searched first and if found it would be replaced by `' known as '`. Note the spaces around some of the entries. They are important, specially for short words, like `' na '` (not available). If the entry would be `'na'` only, each occurrence of `'na'` in the input would be replaced by a single space `' '`. The name `'bernadette'` would then be converted into `'ber dette'`.

The output of the data cleaning routine is a new string where all substrings found in the correction list have been replaced with the corresponding replacement strings. Note that the length of the output string might be different from the input string.

6.3 Step 2: Data Tagging

After an input component string has been cleaned, the next step is to split it at space boundaries into a list of words, numbers and possible separators. The name input `'doctor peter paul miller'` for example is split into a list containing the four words `['doctor', 'peter', 'paul', 'miller']`. All leading and trailing spaces are removed from the list elements.

Using look-up tables and some hard-coded rules each of the list elements is assigned one or more *tags*. The list of possible tags can be found in Appendix B. The hard-coded rules include, for example, tagging an element as a hyphen, a comma, a slash, a number or an alphanumeric word, while most of the other tags (titles, given names, surnames, postcode, locality names, wayfare and unit types, countries, etc.) are assigned to words if they are listed in one of the look-up tables provided. If a word (or a word sequence) is found in a look-up table, it is not only tagged, but it is also replaced by it's corresponding corrected entry in the look-up table. It is possible that a word is listed in more than one look-up table. Consequently, it will be assigned more than one tag (see for example the name word `'peter'` below). Words which are not found in any lookup table and which do not match any of the hard-coded tagging rules are assigned the `'UN'` (unknown) tag.

Table 6.3: Example title look-up table (tag TI).

Original	**Replacement**
'doctor'	'dr'
'doc'	'dr'
'md'	'dr'
'phd'	'dr'
'miss'	'ms'
'misses'	'ms'
'mister'	'mr'

While the input to a tagging routine is a cleaned string, the output is a list of elements and the corresponding list of tags. For the example input name string

```
'doctor peter paul miller'
```

a possible output could be

```
Word list: ['dr', 'peter', 'paul', 'miller']
Tag list:  ['TI', 'GM/SN',   'GM',   'SN'  ]
```

assuming that `'peter'` is listed in both the look-up tables for male given names (`'GM'` tag) and surnames (`'SN'` tag).

Note that a *greedy* matching algorithm is used for the tagging, so that the longest possible sequence of words is matched to the tagging look-up tables when assigning a tag (and substituting corrected values). For example, `'st marys'` is tagged as `'LN'` (locality name) and replaced with the string `'st_marys'`, rather than the `'st'` part of `'st marys'` being tagged as `'WT'` (wayfare type) and being replaced with `'street'`, and `'mary'` being tagged as `'UN'`.

6.4 Step 3: Data Segmenting

Once a word and tag list is available for an input component, a hidden Markov model (HMM) approach is used to segment the input into the possible output fields. The HMM computes the most likely assignment of input elements into output fields. See Chapter 5 for more details.

6.5 Date Cleaning and Standardisation

The cleaning and standardisation of *date* components is undertaken using user-configurable rules. Examples of dates in administrative data sets are date of birth, injury dates, hospital admission dates etc. Dates of birth are often recorded with high accuracy, and they can be checked to some degree if an age field is also available. All routines related to date standardisation are implemented in the module `date.py`.

The aim of date standardisation is to split a given date string into a valid numerical triplet `[day,month,year]`. The date parsing routine consists of an initial cleaning phase, where leading and trailing whitespaces are removed from the input string and various separator strings are replaced by one space.

Date parsing is done using *date format strings*. A format string must consist of three format *directives*, one each for the day, month and year component. The following directives are supported:

`%b` For abbreviated month names (Jan, Feb, Mar, etc.)

`%B` For full month names (January, February, etc.)

`%d` For day of the month as a decimal number between 1 and maximal 31 (depending on the month, and for February if a year is a leap year or not).

`%m` For month as a decimal number between 1 and 12.

`%y` For year without century as a decimal number, i.e. between 00 and 99 (two digits).

`%Y` For year with century as a decimal number (four digits).

While a format string `'%d %m %Y'` matches all dates that start with a day number, followed by a month number and then a four digit year number, a format string `'%b %d %y'` matches for example `'Aug 9 02'`. Date format strings are possible with and without space separators between the directives. If no spaces are given the `%b` and `%B` directives are not possible (only numerical variations of day, month and year are allowed without spaces between them).

A user defines a list of format strings in the `project.py` configuration module. The date parsing routine takes the first format string and tries to parse a given date input string using this format. If it fails, the second format string is tried, and this process is repeated until the given date input string could be parsed, or no more format string are available in the list, in which case an error is returned. Thus, the order in which date format strings are listed in `project.py` is important. The user should order this list of date format strings according to the format in which dates will most likely be represented in the input data sets.

If only a two-digit year has been parsed, it is expanded into a four-digit year using a *pivot year* that is user definable in the configuration module `project.py`. The pivot year separates the range of two-digit years *00-99* into two parts, one that is expanded with *19* (year numbers equal and after the pivot year) the other with *20* (year numbers smaller than the pivot year). For example, if the pivot year is set to `04`, a two-digit year value of `68` is expanded into `1968`, a year value of `04` is expanded into `1904` but a year value of `03` is expanded into `2003`.

6.6 Program 'pyStandard.py'

The program **pyStandard.py** is the main program for data cleaning and standardisation. In its current version, it can read data from text files with various formats and write the cleaned and standardised records back into text files. It is planned to add database access (SQL) capabilities in a future version of this program. Parallel processing capabilities will also be added in a future release.

The program can be called from the command line with the following arguments

python pyStandard.py *[project_module] [first_rec] [num_rec] [options]*

The compulsory arguments are

- *project_module*
  The name of a project configuration module, for example `'project.py'`[1], that contains all configurable options for a certain data standardisation and linkage project. It is important that the file name of this project module has a `'.py'` extension, as otherwise the Python system will not be able to import it. Note that `project.py` is the only Python module in the **Febrl** system which a user needs to modify to configure the system for his or her needs.
- *first_rec*
  The number of the first record to be processed, counting from zero. Because it is assumed that one line in the input file contains one record, a record number always equals the corresponding line number (also counting from zero, otherwise you need to subtract one).
- *num_rec*
  The number of records to be processed.

A number of optional arguments can be given *after* the compulsory arguments. These *options* are:

- `-h`
  Write one header line containing the name of the output fields at the beginning of the output file (first line).
- `-hmm-name` *[file_name]*
  Load and use the hidden Markov model (HMM) with the given file name for the name component standardisation. This option overwrites the default HMM file name given in the configuration module `project.py`. See Chapter 7 for a discussion of how to create HMM files using training data.
- `-hmm-loc` *[file_name]*
  Load and use the hidden Markov model (HMM) with the given file name for the geocode/locality component standardisation. This option overwrites the default HMM file name given in the configuration module `project.py`. See Chapter 7 for a discussion of how to create HMM files using training data.
- `-l` *[file_name]*
  Activate logging and write status, warning and error information into a log file with the given name. It is possible to omit a file name, in which case the default log file name as defined in the configuration module `project.py` is used. Note that if logging is activated, error and warning messages are always written to the log file, while other messages are written according to the setting of the verbosity level.
- `-v1`
  Set verbosity (and logging if activated) to level 1 (medium volume output is printed).
- `-v2`
  Set verbosity (and logging if activated) to level 2 (high volume output is printed).

[1] Although we will refer to the project configuration modules as `project.py` for clarity, we recommend that you copy the `project.py` file and rename the copy something specific to each project, such as `my-project1.py`.

- `-nowarn`
  Suppress the printing[2] of warning messages (but they are still written to the log file if logging is activated).

Although only a small number of settings can be specified using the command line arguments and options, many more can be set in the configuration module `project.py` (see Chapter 10). Prior to data cleaning and standardisation process the user should modify these settings to her or his requirements.

An example invocation of **pyStandard.py** would look like:

```
python pyStandard.py my-project1.py 1 1000000 -v1 -l
```

This would read from an input file and write into an output file as defined in the configuration module `my-project.py`. It would skip the first input record (possibly because it is a header line) and from then on it would clean and standardise one million records. The verbose output is set to level 1, so moderate output is given, and logging is activated so information is written into the log file as defined in `my-project.py` (because no log file name is given).

[2] When we use the terms *print* or *printing* we mean the process of sending text to *stdout* (standard output), which is typically the terminal or console session in which you are working. We do not mean the process of making marks on paper.

CHAPTER

SEVEN

Hidden Markov Model Training

Note: A clearer, more detailed guide to the HMM training process will appear in this chapter in future versions of this manual. However, at this stage the authors are themselves still working out the best and most efficient procedures for training HMMs in order to get the best possible results with the minimum of effort.

Before data cleaning and standardisation can be performed on a new data set, two hidden Markov models (HMMs) – one for the name component and one for the geocode/locality (address) components – need to be trained using training records from the same data set one wants to clean and standardise, or from a similar data set (or data sets).

Training data consists of comma separated sequences with `tag:hmm_state` pairs. Each sequence is a *training example* that is given to the HMM, and the HMM learns the characteristics of a data set by using all training examples that it is given during training. Maximum likelihood estimates (MLEs) for the matrix of transition and observation probabilities (see Chapter 5) for an HMM are derived by accumulating frequency counts of each type of transition and output from the training examples. Because frequency-based MLEs are used, it is important that the records in the training data set(s) are reasonably representative of the overall data set(s) to be standardised. The **pyTagdata.py** module provides various options to automatically choose a random subset of records from a larger input to act as training records. However, the HMMs are quite robust and are not overly troubled if the records in the training data set(s) do not represent an unbiased sample of records from the target data. For example, it is possible to add training records which represent unusual records without degrading the performance of the HMMs on more typical records. HMMs also *degrade gracefully*, in that they still perform well even with records which are formatted in a previously unencountered manner. A simple set of training examples for the name component might look like

```
GF:gname1, SN:sname1
UN:gname1, SN:sname1
GF:gname1, GM:gname2, UN:sname1
GF:gname1, GM:sname1
GF:gname1, UN:gname2, SN:sname1
```

Each line in the example above corresponds to one training record, and contains a sequence that corresponds to a particular path through the various (hidden, unobserved) states of the HMM (the lower-case entities following each colon) together with the corresponding observation symbols (tags). Theses training examples are extracted from the original data set using the **pyTagData.py** program and the HMMs are created using the **pyTrainHMM.py** program. What follows is a basic step-by-step guide for hidden Markov model training.

1. First, create a file with a small number of training records using the **pyTagData.py** program. About 100 records should be enough.

2. Open this file with your favourite text editor. Modify the tagged training records. Comment out lines that have an incorrect tag sequence (add a hash character `'#'` at the beginning of the line). For correct tag sequences, add the appropriate HMM state for each tag of the sequence. Be sure to use lowercase for the state names, and to only use state names listed in Appendix A. Do not leave any spaces between the tag name (in uppercase), the separating colon and the state name (in lowercase). Do not remove the commas which separate each `tag:hmm_state` pair. Only one

training sequence should be activated (that is, not commented out) per input data record. We plan to provide a simple graphical user interface to make this editing task faster in a later version of **Febrl**.

3. Create an initial HMM with **pyTrainHMM.py** using the modified training file. Set the smoothing option `'-s'` to either *laplace* or *absdiscount*. Borkar et al. [2] suggest that absolute discounting seems to work best, but we have also had good results with Laplace smoothing.

4. Create a second, larger training file (with e.g. 1000 records) again using the **pyTagData.py** program this time using the `'-hmm'` option (followed by the file name of the initial HMM file just created). In this way the initial HMM which you created using just 100 training records will be used to *bootstrap* the tagging of this second, larger training file. This reduces the amount of work which needs to be done by the person editing training file, because it is much easier to correct existing states associated with each tag than it is to add states *de novo*, as was necessary in step 2 above.

5. Open the second training file and again manually inspect all training records. Comment out incorrect training records, and modify the one closest to the correct sequence appropriately by changing the HMM state names to whatever is most appropriate. Again, only one training sequence should be activated (not commented out) per input data record.

6. Create a second HMM using the second training file. Set the smoothing options as desired.

7. Create a third training file by reprocessing your second, corrected training file using the **pyTagData.py** program with the `'-hmm'` option (followed by the file name of the second HMM file you just created) plus the `'-retag'` option (followed by the name of the your second, corrected training file). Be sure to specify a different output file name for this third training file so that the second training file which you are reprocessing is not overwritten. Set the smoothing options as desired as previously.

8. Examine this third training file. You will see that records in which the sequence of `tag:hmm_state` pairs has changed will be marked with `'***** Changed'`. Examine only the changed records (since the unchanged records you have already verified in previous steps) and correct whose which appear to be incorrect. Repeat the previous three steps until no more changes are detected in the training file. Note that you also wish to edit the correction lists and look-up tables to improve the observation symbol tagging of the training data (as opposed to the hidden state assignment). You should retag the training file and retrain the HMM after making such modifications to ensure that the changes have not caused other problems, and to ensure that the HMM reflects the latest version of the tagging look-up tables.

9. Finally, in `project.py`, set the default HMM file name to the last HMM file you created. Alternatively you can load a specific HMM when running the standardisation program **pyStandard.py** by using the appropriate option `'-hmm-loc'` and/or `'-hmm-name'` and your HMM file(s).

This training cycle can be enhanced in various ways. For example, once a reasonably good HMM training set has been developed, it can be further improved by adding examples of unusual records to it. By definition, such unusual records occur in the input data only infrequently, and thus very large numbers of training records would need to be examined if they were to be found manually. However, by using the `'-freqs'` option to the **pyTagData.py** module, it is possible to obtain a listing of record formats in ascending order of frequency in a particular input data set (that is, the least common record formats are listed first). Typically quite large numbers of records are specified for processing by **pyTagData.py** when using the the `'-freqs'` option – 200,000 would not be unusual. Of course, there is no prospect of being able to inspect all 200,000 records in the resulting training file, but records with unusual patterns amongst those 200,000 records can be found at or near the top of the output file specified following the `'-freqs'` option. Corrected versions of the `tag:hmm_state` pair sequences for those unusual records can then be added to the smaller, 1000 record training file discussed in the previous paragraphs, and the HMM then retrained using **pyTrainHMM.py**.

7.1 Program 'pyTagData.py'

The program **pyTagData.py** is used to create tagged training records selected from the original data set. Each training record is selected randomly from the input data set, cleaned and tagged in the same way as done in the data cleaning and standardisation program **pyStandard.py**. The tag sequence (or sequences) are written to the training file together with the commented original record.

The program is called from the command line with the following arguments

python pyTagData.py *[project_module] [tag_mode] [train_file] [first_rec] [last_rec] [num_rec] [options]*

The compulsory arguments are

- *project_module*
  The name of a project configuration module, for example `'my-project1.py'`, which is an edited copy of the `project.py` module supplied with **Febrl**. The project configuration module contains all configurable options for a certain data standardisation and linkage project. It is important that the file name of this project module has a `'.py'` extension, as otherwise the Python system will not be able to import it. Note that project configuration module is the only Python module a user has to modify according to her or his needs.

- *tag_mode*
  Setting of the mode which will be used to tag the training records. Possible are either `'name'` or `'locality'`. In the first case, the name component of the training records will be extracted from the input data records and tagged using look-up tables for names, and in the second case the geocode and locality components are extracted and tagged using look-up tables for the geocode and locality components.

- *train_file*
  The name of the output data file which will contain the tagged training records after the **pyTagData.py** has been run. The tag sequences in this output file are non-quoted comma separated (file type `CSV`).

- *first_rec*
  The record number of the beginning of the block from where training records should be selected randomly. It is assumed that one line in the input file corresponds to one record, thus record and line number correspond (counting from zero).

- *last_rec*
  The record number of the end of the block from where training records should be selected randomly.

- *num_rec*
  The number of training records that should be selected randomly in the block between *first_rec* and *last_rec*. The selected records will be cleaned, tagged and their tag sequences will be written to the output file.

A number of optional arguments can be given *after* the compulsory arguments. These *options* are

- `-hmm` *[file_name]*
  Load and use the hidden Markov model (HMM) with the given file name to tag and standardise the selected training records. This allows *bootstrapping* of the training process. If the `'-hmm'` option is used, the training records in the output file will be tagged and annotated with HMM state names.

- `-retag` *[file_name]*
  This option can be used to re-process an existing training file. It can only be used together with the `'-hmm'` option. Note that the values given for *first_rec*, *last_rec* and *num_rec* are overridden when the `'-retag'` option is used (the records in the training file to be re-processed are used instead). This option is useful if one wants to re-tag a training file after look-up tables have been changed (which might result in a different tagging behaviour), or if an HMM has been updated.

- `-freqs` *[file_name]*
  With this option it is possible to compile and write the frequencies of all `tag:hmm_state` pair sequences in ascending order into a file with the given name. This is useful for finding examples of unusual patterns of names or addresses which might need to be added to the training file(s). This option can only be used together with the `'-hmm'` option.

- `-l` *[file_name]*
  Activate logging and write the status, warning and error information into a file with the given name. It is possible to omit a file name, in which case the default log file name as defined in the configuration module `project.py` is used. Note that if logging is activated, error and warning messages are always written to the log file, while other messages are written according to the setting of the verbose level.

- `-v1`
  Set verbose output (and logging if activated) to level 1 (medium volume output is printed).

- `-v2`
  Set verbose output (and logging if activated) to level 2 (high volume output is printed).

- `-nowarn`
  Suppress the printing (to stdout) of warning messages (but they are still written to the log file if logging is activated).

The tagged training records in the file output by **pyTagData.py** can be used to train a hidden Markov model (HMM) with the **pyTrainHMM.py** program. **pyTagData.py** skips through records in the input data file (which is the one defined in `project.py`) until record number *first_rec* and from then on randomly selects *num_rec* records in the block between *first_rec* and *last_rec*. For example, if *first_rec* is set to 0 and *last_rec* is set to the number of records in the data file, *num_rec* records will be selected randomly from the whole data set.

If the option `'-hmm'` followed by the file name of a HMM file is given, the training records are given both tags and (hidden) states, as `tag:hmm_state` pairs, using this HMM. This allows a semi-automatic training process, where the user only has to inspect the output training file and change HMM states for cases that are standardised incorrectly. This mechanism reduces the time needed to create enough records to train a HMM training.

The selected original input records (name or geocode/locality component) are written to the output file as comment lines with a hash `'#'` character and the line number from the input file (starting with zero) at the beginning of a line. After each input data line, one or more lines with tag sequences follows.

The user has to manually inspect the output file and delete (or comment out) all lines with tags that are not correct, and insert a HMM state name for each observation tag in a sequence (or modify the HMM state given).

For example, if we have the three selected input records (name component):

```
'dr peter baxter dea'
'miss monica mitchell meyer'
'phd tim william jones harris'
```

they will be processed (depending on the available look-up tables) and written into the output training file as

```
#0:  |dr peter baxter dea|
  TI:, GM:, GM:, GF:
  TI:, GM:, SN:, GF:
  TI:, GM:, GM:, SN:
  TI:, GM:, SN:, SN:
#1:  |miss monica mitchell meyer|
  TI:, UN:, GM:, SN:
  TI:, UN:, SN:, SN:

#2:  |phd tim william jones harris|
  TI:, GM:, GM:, UN:, SN:
```

If the '-hmm' option is set the output will be something like (again depending on the available look-up tables):

```
# 0:  |dr peter baxter dea|
# TI:titl, GM:gname1, GM:gname2:, GF:sname1:
# TI:titl, GM:gname1, SN:sname1:, GF:sname2:
  TI:titl, GM:gname1, GM:gname2:, SN:sname1:
# TI:titl, GM:gname1, SN:sname1:, SN:sname2:

# 1:  |miss monica mitchell meyer|
  TI:titl, UN:gname1:, GM:sname1, SN:sname2
# TI:titl, UN:gname1:, SN:sname1, SN:sname2

# 2:  |phd tim william jones harris|
  TI:titl, GM:gname1, GM:gname2, UN:sname1, SN:sname2
```

7.2 Program 'pyTrainHMM'

Once tagged training data has been created using the program **pyTagData.py** and edited by a user, a hidden Markov model can be created using **pyTrainHMM.py**.

The program is called from the command line with the following arguments

python pyTrainHMM.py *[project_module] [tag_mode] [train_file] [hmm_file] [options]*

The compulsory arguments are

- *project_module*
  The name of a project configuration module, for example 'my-project1.py', that contains all configurable options for a certain data standardisation and linkage project. It is important that the file name of this project module has a '.py' extension, as otherwise the Python system will not be able to import it. The project configuration module should be created by copying the project.py module supplied with the **Febrl** system and then renaming and editing that copy as appropriate.

- *tag_mode*
  Setting of the mode which will be used to tag the training records. Possible are either 'name' or 'locality'. In the first case, the name component of the training records will be extracted from the input data records and tagged using look-up tables for names, and in the second case the geocode and locality components are extracted and tagged using look-up tables for the geocode and locality components.

- *train_file*
  The name of the input data file containing the tagged training records. The tag sequences in this input file must be non-quoted comma separated (type CSV), as created with **pyTagData.py**.

- *hmm_file*
  The name of the HMM file to be written. The HMM file is a text file containing all the parameters of the HMM, i.e. the state and observation (tag) names; and initial, transition and observation probabilities.

A number of optional arguments can be given *after* the compulsory arguments. These *options* are:

- -s *[smoothing_method]*
  Smoothing of HMM parameters. Smoothing methods implemented are 'laplace' (for the Laplace method) and 'absdiscount' (for the absolute discounting method). Both smoothing methods are described in [2].

- `-l` *[file_name]*
  Activate logging and write status, warning and error information into a file with the given name. It is possible to omit a file name, in which case the default log file name as defined in the project configuration module (which is based on `project.py`) is used. Note that if logging is activated, error and warning messages are always written to the log file, while other messages are written according to the setting of the verbose level.

- `-v1`
  Set verbose output (and logging if activated) to level 1 (medium volume output is printed to stdout).

- `-v2`
  Set verbose output (and logging if activated) to level 2 (high volume output is printed to stdout).

- `-nowarn`
  Suppress the printing of warning messages to stdout (but they are still written to the log file if logging is activated).

This module can be used to train a hidden Markov model (HMM) using tagged data that was created by **pyTagData.py** and then edited manually.

The format of the training data input file must be as follows:

- Comment lines start with a hash character (`'#'`). Blank lines are allowed and are just skipped.
- Each non-empty line that is not a comment line must contain one training record.
- Training records must contain a comma separated sequence of pairs

  ```
  tag:hmm_state
  ```

  where the `tag` is one of the possible tags as listed in Appendix B, and `hmm_state` is one of the possible states from the state lists in Appendix A (either for the name or the geocode/locality components). Any unknown tag or state in the training data will result in an error and the program stops.

CHAPTER

EIGHT

Record Linkage

The record linkage module **pyLinkage.py** is currently under development, and details will be provided in a future version of this manual. Currently the only record linkage components available are simple routines for the probabilistic linkage of *date* fields, as well as modules for *approximate string comparisons* and *name encodings*. These components are all required by the forthcoming **pyLinkage.py** module.

8.1 Date Linkage

Dates like date of birth or hospital admission dates are cleaned in the standardisation process as described in Section 6.5, and dates are then available as numerical triplets `[day,month,year]`.

Comparing two dates can be accomplished in various ways. First, the number of digit transpositions and substitutions can be counted. For example, the two dates `[12,11,1968]` and `[21,11,1969]` have one transposition (the day digits) and one substitution (the last year digit). Substitutions and transpositions occur due to data entry errors, and the user might choose to tolerate a certain number of them in the record linkage process.

A second possibility of comparing dates is to count the number of days by which they differ. Taking the example above, the two dates differ by `374` days. For a given pair of dates, the user might tolerate a certain number of days difference between the pair of dates being compared.

Similar to an absolute day difference is a percentage difference relative to a certain fixed date. Taking again the above example, and setting the fixed date to the time of writing of this text (2 May 2002), the difference between the two dates is `3.16%`. This fixed date can be set in the configuration module `project.py`.

So when computing the linkage weight for a pair of dates, the user might like to tolerate certain errors or differences. Various settings can be configured in `project.py`. For substitutions and transpositions, the maximum number tolerated per day, month and year can be specified. If no transpositions or substitutions occur in a field (day, month or year) between two dates, the corresponding linkage weight is set to the agreement weight as defined in the configuration module. If the number of transpositions or substitutions is smaller or equal to the maximum defined in the configuration module, then the following formula is applied (the example is given for transpositions, but the linkage weights for substitutions, as well as for day and percentage differences are computed in a similar fashion).

$$transposition\_weight = agr\_weight - \left(\frac{num\_of\_trans}{max\_num\_of\_trans + 1}\right) \times (agr\_weight + abs(disagr\_weight))$$

This weight computation is similar to the method used in the **AutoMatch** [12] linkage software.

The final linkage weight for two dates can then be either selected to be the maximum, minimum or a weighted sum of the four weights `substitution_weight`, `transposition_weight`, `absolute_day_difference_weight` or `percentage_difference_weight`. Again, this setting can be selected in the configuration module `project.py`.

8.2 Phonetic Name Encoding

Phonetic name encoding is often used to create blocking variables in the linkage process. Several algorithms for phonetic encoding are implemented in the `encode.py` module. Probably the two most popular algorithms are *Soundex* and *NYSIIS*. A variation of these is *Phonex* [10] which tries to improve the matching quality by preprocessing names before they are encoded. A fourth, more recently developed algorithm is called *Double-Metaphone* [14]. It accounts better for non-English words, like European and Asian names. Similar to *NYSIIS*, *Double-Metaphone* returns a code only consisting of letters, while *Soundex* and *Phonex* return an alpha-numerical code with a fixed length of four. Note that in some cases *Double-Metaphone* returns two codes, according to two different variations in pronunciation. In general, *Double-Metaphone* seems to be closer to the correct pronunciation of names than *NYSIIS*. All of these phonetic codes are particularly sensitive to errors in the first letter of a name. Therefore, the `encode.py` module includes the ability to compute codes from a reversed version of a string.

8.3 Approximate String Comparison

Algorithms for approximate string comparisons are important for good linkage results, as the numerical value they return is used to compute matching weights for string fields like names and addresses. Various algorithms for approximate string comparisons have been developed, in both the medical record linkage [15] and in the computer science and natural language processing communities. In the **Febrl** system, several approximate string comparison algorithms are implemented in the `stringcmp.py`. All string comparison routines in this module return a value between 0.0 (two strings are completely different) and 1.0 (two strings are the same).

The *Jaro* string comparator and its modification due to *Winkler* [15] are commonly used in record linkage software. They compute the number of common characters in two strings, the lengths of both strings, and the number of transpositions to compute a similarity measure between 0.0 and 1.0. The *Winkler* comparator takes into account the fact that typographical errors occur more often towards the end of words, and thus gives an increased value to characters in agreement at the beginning of the strings.

In the *Bigram* algorithm, the number of common bigrams in the two strings is counted and divided by the average number of bigrams in the two strings. Bigrams are the two-character substrings in a word, e.g. `'peter'` contains the bigrams `'pe'`, `'et'`, `'te'` and `'er'`. The *Edit distance* algorithm (also known as the *Levenshtein distance*) counts the minimum number of deletions, transpositions and insertions that have to be made to transform one string into the other.

CHAPTER

NINE

Auxiliary Programs

9.1 Program 'pyRandomSelect'

This simple program reads in a data file and randomly selects records according to the given argument. It writes these records unchanged into the output file.

The program is called from the command line with one of the following argument lists

python pyRandomSelect.py *[in_file]* *[out_file]* `-perc` *[percentage_value]*

or

python pyRandomSelect.py *[in_file]* *[out_file]* `-num` *[num_records]*

The compulsory arguments are

- *in_file*
  Name of the input file with the original data records.
- *out_file*
  Name of the output file where the randomly selected records are written into.
- `-perc` *[percentage_value]*
  Set the percentage of how many records should be selected randomly. The percentage value must be larger than 0.0 and smaller than 100.0.
- `-num` *[num_records]*
  Alternatively, the absolute number of records to be selected randomly can be given as an argument. The value must be positive, and smaller than the total number of records in the input file.

CHAPTER

TEN

Configuration using a Module derived from 'project.py'

Besides the command line arguments and options for the main programs, all configuration options for the **Febrl** system can be specified in one per-project configuration module derived from the `project.py` module which is supplied with the system. As this configuration module is a command line argument itself to all main programs, it is possible to have several, distinct configurations modules with different settings – one per linkage project for example. Because configuration modules (derived from the `project.py` module) are in fact normal Python program modules, it is important that the file name of this module has a `'.py'` file extension, as otherwise the Python system is not able to import it. In addition, it must follow the Python language syntax. Most importantly, comments start with a hash character `'#'`. The module is mainly self-explanatory, in that a comment block explains the meaning and usage of a setting below it.

Therefore, in order to create a new record linkage project, simply make a copy of the `project.py` file supplied with **Febrl**, rename this copy to something appropriate (but ending in `'.py'`) and then edit it to set the various parameters and options as described in the following tables.

Note: Wherever a file name needs to be specified, either in the project configuration module or as a command line argument, you may specify

- just a file name, in which case it will be read from or written to the directory in which the relevant **Febrl** program is being executed (that is, the current or present working directory);
- a file name qualified by a relative path, in which case the path is relative to the directory in which the relevant **Febrl** program is being executed (that is, the current or present working directory);
- a fully-qualified path name (which could even be a network path name).

Table 10.1: Verbose output and logging options.

Option	Description
`verbose`	Verbose output level. Possible values are: `0` No verbose output `1` Moderately verbose output `2` Excessively verbose output
`logging`	Write logging information to file. Possible values are: `0` No logging to file `1` Activate logging to file
`log_file`	Name of the default log file.
`nowarn`	Enable or suppress printing (to stdout) of warning messages. Possible values are: `0` Do print warning messages `1` Do not print warning messages Note that warning messages are still written to the log file if logging is activated.
`proc_ind`	Number of records between a process indication message is printed. Set to `-1` to disable process indication messages.

Table 10.2: Input and output file options.

Option	Description
`in_file_name`	Name of the input file containing the original data.
`in_file_type`	Type of the input data file. Possible file types are: `CSV` Comma separated values, fields separated by commas `TAB` Tabulator separated values, fields separated by commas `COL` Column wise, fields within specific column ranges
`out_file_name`	Name of the output file to which the standardised data will be written.
`out_file_type`	Type of the output data file. Possible file types are: `CSV` Comma separated values, fields separated by commas `CSVQ` Comma separated values, where each field starts and ends with a quote character `TAB` Tabulator separated values, fields separated by commas `TABQ` Tabulator separated values, where each field starts and ends with a quote character `COL` Column wise, fields within specific column ranges

Table 10.3: Input field options.

Option	Description
input_component	This is a Python dictionary that contains information on which field(s) or which columns from an input record should be assigned to which component (name, geocode, locality, date1 or date2). If a component is not available, set the corresponding entry to an empty list []. For the name component, if given names and surnames are already available in two separated input fields, it is possible to define the givenname and surname components separately. In such a case the name component needs to be set to an empty list []. If the input data file is comma or tabulator separated (input file type is CSV or TAB) each component is specified as a list of one or more field numbers (starting with 0). For column oriented input data files (input file type COL), a list of column ranges (start,length) needs to be specified for each component.
input_space_sep	This Python dictionary contains a flag for each input component. If set to 1, a space character ' ' is inserted between the fields of a component before they are concatenated.
input_check_spilling	This Python dictionary contains a flag for each input component. If set to 1, word *spilling* between input fields will be checked and corrected if possible.

Table 10.4: Output field options.

Option	Description
output_field	This is a Python dictionary that contains information on which output fields will be written into the output data file and in what order. Similar to the definition of the input components, for each output field a field number (for comma and tabulator separated files) or a (start,length) column range (for column oriented output files) can be given. An empty list [] for an output fields means it will not be written into the output file. Column and field numbers start with 0. Two special fields are name_hmm_proba and geoloc_hmm_proba which are the probabilities returned by the Viterbi algorithm for the most likely HMM state sequence (which is the one chosen for the standardisation). The record number for each processed record can be added to the output by using the special field record_id. Record numbers are counted from the beginning of the input file, starting with zero. Each line in the input file is counted as one record. It is possible to write parts or all of the original input record unmodified into the output file by using the special output field original_input. Three forms of original_input are possible (see the comments in the project.py module for more details): – original_input[in_field_num] for comma and tabulator separated input files. – original_input[in_start_col,in_end_col] for column wise input files. – original_input for the whole input record (the original input line)
output_quote_character	The quote character for quoted output data files (output file type CSVQ or TABQ).

Table 10.5: Look-up table and correction list files options.

Option	Description
name_corr_list_file	Name of the correction list file used for cleaning the name component. See Section 11.1 for more information on the format of this file.
geoloc_corr_list_file	Name of the correction list file used for cleaning the geocode and locality components. See Section 11.1 for more information on the format of this file.
name_lookup_table_files	A list of file names which will be used to tag and correct words in the name component. See Section 11.2 for more information on the format of these files.
geoloc_lookup_table_files	A list of file names which will be used to tag and correct words in the geocode and locality components. See Section 11.2 for more information on the format of these files.

Table 10.6: Name component options.

Option	Description
name_standard_method	The method for name standardisation can either be 'rules' or 'hmm'. A hidden Markov model needs to be available if the 'hmm' method is chosen.
name_hmm_file_name	The name of the default hidden Markov model file that is used for name standardisation. An alternative HMM file can be loaded using the '-hmm-name' option in **pyStandard.py**
name_female_title	Female title words used to determine a gender guess.
name_male_title	Male title words used to determine a gender guess.

Table 10.7: Geocode and locality component options.

Option	Description
geoloc_standard_method	The method for geocode and locality standardisation. Currently only a hidden Markov model based approach is possible, so the value of this option has to be set to 'hmm'
geoloc_hmm_file_name	The name of the default hidden Markov model file that is used for geocode and locality standardisation. An alternative HMM file can be loaded using the '-hmm-loc' option in **pyStandard.py**

Table 10.8: Date component options.

Option	Description
date_parse_formats	A list of date format strings as discussed in Section 6.5.
date_pivot_year	Pivot year for expansion of two-digit year numbers into four-digit year numbers. See Section 6.5 for more details and examples.
date_perc_fix_date	The date relative to which the percentage comparison of two dates is carried out. It can either be a date string in a valid format or the string 'today' in which case the current date is taken.
date_age_fix_date	The date relative to which the age computation of a date is carried out. It can either be a date string in a valid format or the string 'today' in which case the current date is taken.
date_day_m_prob	Probability that days are the same in a linked pair.
date_day_u_prob	Probability that days are the same in an unlinked pair.
date_month_m_prob	Probability that months are the same in a linked pair.
date_month_u_prob	Probability that months are the same in an unlinked pair.
date_year_m_prob	Probability that years are the same in a linked pair.
date_year_u_prob	Probability that years are the same in an unlinked pair.
date_comp_max_subst	Maximum allowed number of tolerated substitutions in date comparison. Must be a numerical triplet for [day,month,year].
date_comp_max_trans	Maximum allowed number of tolerated transpositions in date comparison. Must be a numerical triplet for [day,month,year].
date_comp_max_day_before	Maximum number of days tolerated for the first date being before the second date in a date comparison.
date_comp_max_day_after	Maximum number of days tolerated for the first date being after the second date in a date comparison.
date_comp_max_perc_before	Maximum percentage value tolerated for the first date being before the second date in a date comparison.
date_comp_max_perc_after	Maximum percentage value tolerated for the first date being after the second date in a date comparison.
date_linkage_weight_comp	Final date linkage weight computation, which can be the minimum, maximum or a combination of the four weights: - substitution_weight - transposition_weight - absolute_day_difference_weight - percentage_difference_weight See Section 6.5 for more details. The value of this option can be either 'min', 'max' or a list with four numerical fractional weights that must sum up to 1.0, e.g. [0.1,0.2,0.3,0.4].

CHAPTER

ELEVEN

Look-up and Frequency Files

In the data cleaning and standardisation process, correction lists and look-up tables with word corrections and expansions are needed, and in the linkage process frequency tables can be used to compute matching weight probabilities for various components of names and addresses[1]. These lists and tables are stored in text files and can be edited or created by the user. Three different types of look-up and frequency file are used.

- Correction lists
  These files contain strings (characters or words) and their replacements. They are used in the initial cleaning step in the data standardisation process.
- Look-up tables
  Similar to correction lists, these files contain strings and their replacement. Additionally, groups of table entries are assigned a tag, which is then used in the tagging step. Look-up tables are mainly used for names and addresses.
- Frequency tables
  These tables contain words and integer numbers that corresponds to the frequency of the corresponding word.

Common to all files is that lines starting with the hash character `'#'` are comment lines, and their content is skipped. Note that if a `'#'` character is not at the beginning of a line it is not interpreted as the start of a comment. Instead it will be used as a normal part of a list or table entry.

Note: As frequency tables are only used in the linkage process but not in the data cleaning and standardisation process, no example frequency files are supplied in this version of the **Febrl** software because the linkage modules is not yet included.

11.1 Correction List File Format

Correction list files contain characters or strings and their corresponding corrections (replacements). They are converted into Python lists which are used in the initial data cleaning step to replace a character or string with the corresponding replacement. Correction list files have a file extension `'.lst'`. The format of these files is as follows:

- An entry in the correction-list file is of the form

  $$replacement \texttt{ := } values$$

  where *values* is a list of one or more comma separated strings. Each *value* in this list is replaced with the *replacement* character or string on the left hand-side of the entry.
- All *replacement* and *value* strings have to be quoted, either with single or double quotes.

[1] The **AutoMatch** as formerly sold by *MatchWare Technologies* derived quite small frequency weight tables directly from the input files. The **Febrl** system will allow much larger frequency weight tables derived from external sources (such as a telephone directory) to be used, hence the need to specify the format of the frequency table files.

- An entry can be longer than one line, in which case the second and following lines consist of the *values* list only.
- Command lines are lines that start with a '#' character.

The following example is taken from the 'name_corr.lst' file:

```
# ------------------------------------------------------------------
# Remove characters and words from input (replace with single space)
             ' ' := '.', '?', '~', '_', ':', ';', '^', '=', ' na ',
                    ' n/a ', ' n.a.', '\', ' also ', ' name ',
                    ' only ', ' abbrev ', ' initials ', ' unk ',
                    ' unkn ', ' missing ', ' unknown '

# Correct words and symbols
          ' and ' := '+', '&'
         ' baby ' := ' babe '
      ' baby of ' := ' babyof ', ' babeof ', ' b/o ', ' b.o.'
     ' known as ' := ' knownas ', ' a.k.a. '

# Remove ' from o'brian etc
             ' o' := " o'"
             ' a' := " a'"
# ------------------------------------------------------------------
```

All values in the first entry (an entry that goes over four lines) are replaced with a single space ' '. It is important that, for example, the value ' na ' starts with a space and ends with a space. Assuming these spaces were omitted, i.e. the value would be 'na', then each occurrence of the string 'na' in the input would be replaced by a single space. The word 'annabella' would thus be replaced with 'an bella' which is not what is wanted.

The list of *value* and *replacement* pairs is internally sorted with decreasing length of the *values*. Long *value* strings are therefore replaced before shorter strings or characters are replaced. In this way, the value ' a.k.a. ' is replaced with ' known as ' before full-stops (periods) '.' are replaced by a space ' '.

11.2 Look-up Table File Format

A look-up table file contains one or more blocks of entires, with all entries in a block are being assigned the same tag. Look-up table files have a file extension '.tbl'. The format of these files is as follows:

- A block starts with a line that contains a *tag assignment* with a tag in brackets:

  tag=<*tag*>

  This tag assignment must be written at the beginning of a line. It is possible to have a comment (starting with a '#') after the assignment.
- All following lines are assumed to contain the entries to be tagged with the currently assigned tag, until a line with a new tag assignment is encountered.
- Each entry in a block is of the form:

  key : *values*

  where *values* is a list of none, one or more comma separated strings (not quoted), and *key* (not quoted) is one or several words.
- Each of the *values* in the list will be replaced with the *key* if they are found in an input record.
- If the *values* list is empty, then only the *key* itself will be inserted into the look-up table.

- A *value* can consist of more than one word (spaces between words are possible).
- If a *value* occurs in more than one block and it is replaced with the same *key* but with different tags, all tags are kept and stored in a list for this *value*.
- If a *value* occurs in more than one block and it is replaced with different *keys*, an error message is printed and the program stops. The user then manually has to correct this error.
- Command lines are lines that start with a '#' character.

The following examples are extracted from the 'name_misc.tbl' and 'territory.tbl' files:

```
# ----------------------------------------------------------------
tag=<SP>  # Tag for separator elements
              and :
               or :
         known as : kn as, kn, known

tag=<BO>  # Tag for 'baby of' and similar sequences
             baby :
          baby of :
         daughter :
      daughter of :
              son :
           son of :

tag=<NE>  # Tag for word 'nee' (born as) or surname or givenname (?)
              nee :

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

tag=<TR>  # Tag for territory words

other territories : o/t, o t, other territory, other terr

  new south wales : n s w, new s w, new south w, nsw, n south w,
                    n south wales, new south wa, n south wa,
                    new s wa, n s wales, new s wales

        queensland : q l d, q land, queen land, queens land, qld,
                     queenland

  south australia : s a, s australia, s australian, sa,
                    south australian, southern australia,
                    southern australian

         victoria : vi, vict, vic

western australia : w australia, w australian, wa,
                    western australian, west australia,
                    west australian

# ----------------------------------------------------------------
```

11.3 Frequency-Table File Format

The third type of look-up table files are lists of words with corresponding frequency counts. These files contain two columns separated by a comma, with the first column containing words and the second column containing the corresponding frequency counts (positive integer numbers). These files have a file extension '.csv' (comma separated values).

A probability distribution for a given frequency look-up table is computed internally after loading such a file by summing up all the frequency counts and then dividing each frequency count by this sum.

CHAPTER

TWELVE

Installation

Before you install this software, you need to have Python version 2.2.1 (or later) installed on your system. You can download the Python source as well as binary distributions for various platforms from

http://www.python.org/download

Follow the configuration and installation instructions given on the Python Web site or which come with the Python distribution you have downloaded. Make sure you set the path variables on your system so that you can start Python by simply typing **python** in a command line session (e.g. at the Windows MS-DOS or *secured command* prompt, or in the UNIX or Linux shell prompt).

To do so on Windows systems, you may have to add a **'set PATH='** line to your 'Autoexec.bat' file. Assume you have installed Python on drive `C:` in the directory `Python22` you may have to add

```
set PATH=C:\Python22;\%PATH\%
```

On UNIX systems you may have to update you '.cshrc' or '.bashrc' file by adding the path to your Python installation.

The three main programs in the **Febrl** package are **pyStandad.py** for data cleaning and standardisation, **pyTagData.py** to create tagged training records for hidden Markov models (HMMs), and **pyTrainHMM.py** to train and save HMMs into files so they can be used by **pyStandad.py**.

For the current release, just unzip or untar the **Febrl** distribution file in a convenient location. Be sure to specify the *create directories* option in your unzip utility. On UNIX or Linux systems, you would generally type

```
tar xvfz Febrl.tar.gz
```

or similar.

After unzipping or untarring the **Febrl** distribution, change to the `Febrl` directory and run the various **Febrl** programs from there. Make a copy of the `project.py` module and modify this copy according to your data set(s). That's it at this stage.

Note: Future versions of **Febrl** will use the standard Python `distutils` module to install the various **Febrl** components in the *site-packages* directory, as well as command line wrappers in an appropriate executable directory somewhere on the system path. This will allow a great deal more flexibility, but requires the refactoring of the current code into a slightly more object-oriented form as described in Appendix D.

Hidden Markov Model States

The following two lists contain all possible states for the name and geocode/locality hidden Markov models, respectively.

Table A.1: States for Name HMM

State	Description
titl	Title state
baby	State for *baby of*, *son of* or *daughter of*
knwn	State for *known as*
andor	State for *and* or *or*
gname1	Given name state 1
gname2	Given name state 2
ghyph	Given name hyphen state
gopbr	Given name opening bracket state
gclbr	Given name closing bracket state
agname1	Alternative given name state 1
agname2	Alternative given name state 2
coma	State for comma
sname1	Surname state 1
sname2	Surname state 2
shyph	Surname hyphen state
sopbr	Surname opening bracket state
sclbr	Surname closing bracket state
asname1	Alternative surname state 1
asname2	Alternative surname state 2
pref1	Name prefix state 1
pref2	Name prefix state 2
rubb	Rubbish state, for elements to be thrown away

Table A.2: States for Geocode/Locality HMM

State	Description
wfnu	Wayfare number state
wfna1	Wayfare name state 1
wfna2	Wayfare name state 2
wfql	Wayfare qualifier state
wfty	Wayfare type state
unnu	Unit number state
unty	Unit type state
prna1	Property name state 1
prna2	Property name state 2
inna1	Institution name state 1
inna2	Institution name state 2
inty	Institution type state
panu	Postal address number state
paty	Postal address type state
hyph	State for hyphen
sla	State for slash
coma	State for comma
opbr	Opening bracket state
clbr	Closing bracket state
loc1	Locality name state 1
loc2	Locality name state 2
locql	Locality qualifier state
pc	Postcode state
ter1	Territory name state 1
ter2	Territory name state 2
cntr1	Country name state 1
cntr2	Country name state 2
rubb	Rubbish state, for elements to be thrown away

List of Tags

The following list contains all possible tags for the name and geocode/locality component, respectively.

Tag	Description	Name	Geocode/Locality
TI	Tag for title words	Yes	–
GF	Tag for female given names	Yes	–
GM	Tag for male given names	Yes	–
SN	Tag for surnames	Yes	–
II	Tag for one-letter words (initials)	Yes	–
PR	Tag for name prefix words (like *de*, *la*, *van*, etc.)	Yes	–
ST	Tag for saint words	Yes	Yes
NE	Tag for the word *nee*, which can be a surname but may also mean *born* (in which case it becomes a separator)	Yes	–
BO	Tag for *baby of*, *daughter of* and *son of* sequences	Yes	–
SP	Tag for a separator, like *known as*	Yes	–
PC	Tag for postcodes	–	Yes
CR	Tag for country words	–	Yes
TR	Tag for territory (state) words	–	Yes
LN	Tag for locality name words	–	Yes
LQ	Tag for locality qualifier words	–	Yes
IN	Tag for institution name words	–	Yes
IT	Tag for institution type words	–	Yes
WT	Tag for wayfare type words	–	Yes
WN	Tag for wayfare name words	–	Yes
UT	Tag for unit type words	–	Yes
PA	Tag for postal address type words	–	Yes
VB	Tag for vertical bars (which are the processed form of various brackets and quotes)	Yes	Yes
HY	Tag for hyphens	Yes	Yes
CO	Tag for commas	Yes	Yes
SL	Tag for slashes	–	Yes
NU	Tag for numbers (all numbers in names, but only numbers that do not have 4-digits in geocode/locality	Yes	Yes
N4	Tag for four-digit numbers (that are not listed in the postcode look-up table)	–	Yes
AN	Tag for alphanumeric words, i.e. words that contains both letters and digits	Yes	Yes
UN	Tag for unknown words (i.e. words not listed in any look-up table)	Yes	Yes
RU	Rubbish tag (i.e. words that will be removed from the input)	Yes	Yes

Manifest

The follow files are provided with the current distribution of **Febrl**.

- The main directory contains the Python programs, license files and documentation. Note that a compressed (gzipped) PostScript version of the manual is available for download from the **Febrl** web site but is not included in the standard distribution due to its large size.

```
ANUOS_v1.0.txt
README.txt
LICENSE.txt
config.py
date.py
encode.py
febrldoc.pdf
inout.py
locality.py
mymath.py
name.py
project.py
prRandomSelect.py
pyStandard.py
pyTagData.py
pyTrainHMM.py
simplehmm.py
stringcmp.py
tcsv.py
```

- The `hmm` directory contains some example hidden Markov model training data sets (`'.csv'` files) and some example HMMs derived from them. The training data has been derived from files of NSW death certificates and MDC (Midwives Data Collection) data. It should work adequately with most Australian name data and NSW address data. The tagging look-up tables will need to modified to suit other states of Australia or other countries. In future versions we plan to include look-up tables and example training sets which are suitable for initial use anywhere in Australia. We are also happy to include example files for other countries if these are contributed.

```
./hmm/geoloc-absdiscount.hmm
./hmm/geoloc-laplace.hmm
./hmm/geoloc.hmm
./hmm/geoloc-sample-training-data.csv
./hmm/hmm-states.txt
./hmm/name-absdiscount.hmm
./hmm/name-laplace.hmm
./hmm/name-sample-training-data.csv
./hmm/name.hmm
```

- The `data` directory contains look-up tables, correction-lists and frequency-tables.

```
./data/country.tbl
./data/geoloc_corr.lst
./data/geoloc_misc.tbl
./data/geoloc_qual.tbl
./data/givenname_f.tbl
./data/givenname_m.tbl
./data/institution_type.tbl
./data/locality_name_act.tbl
./data/locality_name_nsw.tbl
./data/name_corr.lst
./data/name_misc.tbl
./data/name_prefix.tbl
./data/post_address.tbl
./data/postcode_act.tbl
./data/postcode_nsw.tbl
./data/saints.tbl
./data/surname.tbl
./data/territory.tbl
./data/title.tbl
./data/unit_type.tbl
./data/wayfare_type.tbl
```

To-Do: Outstanding Development Tasks and Possible Additions and Enhancements

This is an incomplete list of outstanding development tasks and possible additions and enhancements to the **Febrl** system.

- Finish the `pyLinkage`.py module!
- Repackage the code so it can be installed using the standard Python `distutils` module.
- Add a single letter tag for address standardisation, similar to the `II` (initial) tag used in name standardisation. This will allow HMMs to more easily classify single-letter words correctly, and then allow their transformation at the output stage. For example, it is hazardous to transform every instance of the string `' r '` into `' road '` during the initial cleaning stage, but it is quite safe to transform any instance of `'r'` into `'road'` in the output stage where those instances have been classified as the wayfare type.
- Check when using the `'-retag'` option on the **pyTagData.py** module that the training file being reprocessed is not accidentally overwritten.
- Current input data is split into words at space boundaries. It might be useful to allow word splits at other, user-configurable boundaries (at least all whitespace characters).
- Provide a complete set of example files, including data files (not just training files and look-up tables). The difficulty here is that for confidentiality reasons it would be necessary to use synthesised data, or data provided with consent by data donors. Either are quite hard to assemble, but we should be able to create at least some small example data files which can be standardised and linked.
- Modify `pyStandard.py` and `pyTagData.py` so that it is not necessary to know the total number of records in an input file if you want to process the entire file - the program should determine this itself (as `pyRandomSelect.py` already does).
- Provide a simple user interface to the HMM training module to make the checking of training data sets faster and easier. At the moment, these training data sets need to be edited in a text editor and a lot of time and mental energy is wasted scrolling the cursor to the correct place in the file to add or correct states. It is anticipated that the `curses` module will be used to provide a text-mode user interface on UNIX and Linux platforms, and the `Tkinter` module will be used to provide a graphical user interface on all platforms which support this (including Microsoft Windows and Apple Macintosh).
- *Soft code* the observation tags and hidden states used by the HMM standardiser so that users can specify their own sets of tags and states without having to modify the Python code.
- Re-factor the program code to make it a bit more object-oriented, e.g. implement each project and each HMM as a class instance. Doing this should make it easier to integrate **Febrl** into other systems and will also make parallelisation using `PyRO` (see http://pyro.sourceforge.net) easier. See below for further discussion of this.

- Parallelise all modules. Ole Nielsen's `PyPar` module (see http://datamining.anu.edu.au/~ole/pypar/) for accessing MPI (Message Passing Interface) from Python will be used for communication between computing nodes on dedicated clusters and SMP machines, and the `PyRO` module (see above) will be used for communications between more loosely coupled COWs (clusters of workstations).

- Add unit tests to all modules and programs in the **Febrl** systems, using the Python `doctest` facility in the `docutils` module, as well as the standard Python `unittest` module. Of course, unit test will require example data sets to process so that the results can be compared against the expected results (see above).

- Allow more than one standardisation HMM to be trained and used, and use the *forward algorithm* to choose the best HMM for each input record. For example, input data may contain a mixture of addresses from different countries, each of which has quite different conventions for specifying address, each requiring a different HMM.

- Continue to develop the example standardisation training data and tagging look-up tables so that they work well with a wide range of typical Australian name and address data. Accept tagging and training data for other countries from contributors provided the tagging look-up tables are free from copyright restrictions.

- Explore the utility of the *Baum-Welch* EM (expectation maximisation) algorithm in optimising the probabilities in the HMM used for standardisation. Probably use *LogiLab's* fast C code implementation of this (see http://www.logilab.org/hmm/).

- Explore the use of the *forward-backwards* algorithm for developing HMMs without explicitly specifying the hidden states.

- Ensure that the system can process Unicode strings correctly. Note that Python (since version 1.6) has facilities for working with Unicode data, but the current **Febrl** program code has not been written with Unicode strings in mind and may need to be modified. Further to this, it may be possible to implement transliteration tables so that it is possible to link data sets encoded in different alphabets, e.g. in Khmer and Roman alphabets.

- Use Python's `ConfigParser` module for reading in configuration settings. This means a re-design of `project.py`. Alternatively, design a GUI for configuration and save configuration settings as XML files.

- Add *fuzzy* look-up of words in the tagging lists so that the user does not have to specify all possible mis-spelling of abbreviations of a word or group of words for it to be tagged correctly (although the current ability to do that is very useful). It is quite tricky to do a fuzzy lookup efficiently, but worth considering for the next version. Real-life testing has shown that words with minor mis-spellings which should have been tagged as a locality, institution type etc. occur very commonly. However, the HMM still manages to get the output state for most of them correct, but would do even better if they were tagged correctly in the first place by a fuzzy lookup of the tagging tables.

- Add the ability to specify output transformation rules for the output fields, e.g. if the `unit_number` output field starts with `'unit'` (as in, for example, `'unit42'`, then remove the `'unit'` prefix from the `unit_number` field and make the `unit_type` field equal to `'unit'`. There is no need to invent a new rule specification language to implement this – Python is already easy enough for users of **Febrl** to write their own transformation rules. All the user needs to do is specify a function name for the transformation of the output field. However, some more object-oriented refactoring would make this easier to implement, e.g. if the output was an object (class instance) which could be passed to such transformation functions.

- Extend the above idea to input processing as well, to allow more than just substitution.

- Add the ability to read and write data from and to SQL and ODBC databases and data sources. Also the ability to read and write data as XML documents.

- Add the ability to read multi-line and maybe even hierarchical-format files (since hierarchical databases are still used by a lot of mainframe data processing systems written in COBOL etc.). The ability to writing these formats is another matter, however.

- Provide an improved, more robust version of the `tcsv.py` delimited file parsing module.
- Notes on object orientation of **Febrl**

  1. Make a `Project` class, with all the options as they now are in the `project.py` file as attributes of the class, with default values specified in the `Project` class definition.
  2. To override the defaults for a particular project, the user can just specify the values when instantiating the project, or set them afterwards. For example the first part of a project file would look like this:

     ```
     from febrl import *
     myproject = Project(name="MDC linkage 2002",
                         infile="mdc2002.csv",
                         outfile="mdc2002_out.csv",
                         etc.)
     ```

     where the names and values of the formats are exactly as they are now. Or like this:

     ```
     from febrl import *
     myproject = Project()
     myproject.name="MDC linkage 2002"
     myproject.infile="mdc2002.csv"
     myproject.outfile="mdc2002_out.csv"
     ```

  3. If users want to permanently override the defaults for their projects, they just subclass the `Project` class and override the attributes they want to change:

     ```
     class TimProject(Project):
       self.filetype='SQL'

     myproject=TimProject()

     etc.
     ```

  4. Make **pyTrainHMM.py**, **pyTagData.py**, **pyStandard.py**, etc. methods of the `Project` class, so to run them, you invoke them like this:

     ```
     myproject.TagData()
     ```

     The parameters for `TagData()` are all obtained from the attributes of the `myproject` instance of the `Project` class. Or some could be overridden in the method call:

     ```
     myproject.TagData(logging=1)
     ```

  5. We add methods so that projects know how to print their own configurations in a neat format, i.e. they write their own documentation.
  6. An entire project can then be run from a single script, and users just comment out the bits the don't want to run at a particular time. This leaves a much better audit trail than command-line programs and makes it easier to re-run things at a later date, for example:

     ```
     from febrl import *
     myproject = Project(name="MDC linkage 2002",
                         infile="mdc2002.csv",
                         outfile="mdc2002_out.csv",
                         etc.)
     # myproject.TagData(logging=1)
     myproject.TrainHMM()
     # myproject.Standardise()
     ```

  7. `.Save()` and `.Load()` methods should be added to the `Project` class, and these serialise and de-serialise all the attributes etc of the project. This would allow projects to be set up, run and saved to a file, and reloaded interactively from the Python prompt as well as from script files as above.

8. Naturally, the project log, the HMMs etc. all become attributes of each project, and they are serialised and deserialised with the rest of the project. Methods are provided to print them out in various formats so you can examine them.
9. We can still provide a command-line interface to all this using wrapper functions around the `Project` class methods, so it will be possible to maintain backwards compatibility with Version 0.1 if we want to, or if some people prefer a command-line approach.
10. All these changes do not involve a major re-write – mostly just changing functions into methods. And it can be done slowly, bit by bit, not all at once.

- Currently **Febrl** is oriented towards batch processing using modules invoked from the command line (or from a batch file or shell script). This is probably the most useful interface for biomedical researchers. However, later versions may offer other APIs, such as an object-oriented Python API and Web service interfaces (via XML-RPC, SOAP, HL-7 or CorbaMed), in order to facilitate the embedding of **Febrl** in other systems such as cancer registry databases or even Patient Master Indexes (PMIs). The C language version of the Python language can itself be quite easily (and freely) embedded in other software. Although we haven't tried it, **Febrl** should work OK under Jython, the Java implementation of the Python language, making it easy to embed in Java-based systems.

- Explore the usage of the `Optik` module for command line argument parsing. See the Python Getopt SIG web site at http://www.python.org/sigs/getopt-sig/.

- It is not always sensible to concatenate fields which are already quite well segmented in a database record, only to try to parse them out again. One way not to lose such pre-existing segmentation without having to modify **Febrl** very much would be to add an option to add a comma (or some other character) as a delimiter between fields in the concatenated name or address string which is then presented to the **Febrl** parser. Such a delimiter is not wanted between every field, so there would need to be some way of specifying where to insert it. For example, if the original record was:

```
Field1 = Wayfare number
Field2= Wayfare name and type
Field3= Locality
Field4=State/Territory
Field5=Postcode
```

  then it would be useful to be able to re-concatenate that as:

```
Field1 Field2, Field3 Field4 Field5
```

  and present that to the parser. Of course, the HMMs would have to be trained using data in that format (i.e. with comma interpolation between fields 2 and 3 switched on), but at least we would then be able to distinguish between:

```
23 Smith St North, Fairfield NSW 2345
```

  and

```
23 Smith St, North Fairfield NSW 2345
```

  At the moment, this is presented to the standardiser as:

```
23 Smith St North Fairfield NSW 2345
```

  Another idea is to use a dictionary of wayfare names in each locality, derived from the phone book or similar (such as that LPI geocoding database) to distinguish property names and wayfare names, e.g.

```
Wymallee Arthur St Gundagai NSW 2345
Windy Willows Ave Littleville NSW 2345
```

Now we know that `'Wymallee'` is the property name (because it sounds like one), and `'Arthur'` is the wayfare name (also, `'Wymalle Arthur'` would be a strange name for a street), but for the second address, `''Windy Willows'` is the wayfare name and there is no property name. Now, if it were possible to look up all streets in e.g. Gundagai, and a match was found for `'Arthur'`, but not `'Wymallee Arthur'`, then it is possible to infer, as a deterministic post-processing rule, that `'Wymallee'` should be re-assigned to the property name output field. All this sort of processing, which is domain and site-specific, can just be done by users writing their own Python functions or methods to post-process the standardised data – like a plug-in. So all we need to do is provide a hook for such plug-ins, and a few examples.

- Make use of the `import` internals functions to load a user-specified module (e.g. `myproject.py`), instead of the `exec(...)` code currently being used.
  See: http://www.python.org/doc/current/lib/module-imp.html

Version History

0.1 First public release (06 September 2002)

0.1.1 Several updates in Appendix D (ToDo list) (xx September 2002)

Support Arrangements

As stated in the License (see Appendix G), **Febrl** is provided on an *AS IS* basis, *WITHOUT WARRANTY OF ANY KIND*, either expressed or implied.

However, the authors are keen to learn of any bugs, defects or limitations in the software. At this stage, just e-mail the details to the authors. A formal mailing list for **Febrl** will be set up in due course.

Users or potential users of **Febrl** should note that although problems with the programs will be attended to on a *best effort* basis, the authors have other responsibilities (not to mention families and friends) and do not undertake to resolve problems within any particular time frame, or at all, as the case may be. Because the source code for **Febrl** is freely available, users may be able to resolve many problems themselves, or with the help of others who have experience with the Python programming language. We would appreciate it if copies of such fixes and modifications are e-mailed to us so that they can be incorporated into future versions of **Febrl**.

ANU – Open Source License

AUSTRALIAN NATIONAL UNIVERSITY OPEN SOURCE LICENSE (ANUOS LICENSE) VERSION 1.0

1. DEFINITIONS

Associated Documentation and Data Files shall mean all files distributed in conjunction with the *Original Software* which are not computer program code.

Commercial Use shall mean distribution or otherwise making the *Covered Software* available to a third party.

Contributor shall mean each entity that creates or contributes to the creation of *Modifications*.

Contributor Version shall mean in case of any *Contributor* the combination of the *Original Software*, prior *Modifications* used by a *Contributor*, and the *Modifications* made by that particular *Contributor* and in case of the *Australian National University* in addition the *Original Software* in any form, including the form as *Executable*.

Covered Software shall mean the *Original Software* and *Associated Documentation and Data Files* or *Modifications* or the combination of the *Original Software* and *Associated Documentation and Data Files* and *Modifications*, in each case including portions thereof.

Electronic Distribution Mechanism shall mean a mechanism generally accepted in the software development community for the electronic transfer of data.

Executable shall mean *Covered Software* in any form other than *Source Code*.

Initial Developer shall mean the individual or entity identified as the *Initial Developer* in the *Source Code* notice required by *Exhibit A*.

The Australian National University shall mean the *Australian National University, ABN 52-234-063-906*, a body corporate pursuant to the *Australian National University Act 1991* of Canberra, in the Australian Capital Territory.

Larger Work shall mean a work, which combines *Covered Software* or portions thereof with code not governed by the terms of this *License*.

License shall mean this document.

Licensable shall mean having the right to grant, to the maximum extent possible, whether at the time of the initial grant or subsequently acquired, any and all of the rights conveyed herein.

Modifications shall mean any addition to or deletion from the substance or structure of either the *Original Software*, the *Associated Documentation and Data Files* or any previous *Modifications*. When *Covered Software* is released as a series of files, a *Modification* is:
a) Any addition to or deletion from the contents of a file containing *Original Software*, *Associated Documentation and Data Files* or previous *Modifications*.
b) Any new file that contains any part of the *Original Software*, *Associated Documentation and Data Files* or previous *Modifications*.

Original Software shall mean the *Source Code* of computer software code which is described in the *Source Code* notice required by *Exhibit A* as *Original Software*, and which, at the time of its release under this *License* is not already *Covered Software* governed by this *License*.

Patent Claims shall mean any patent claim(s), now owned or hereafter acquired, including without limitation, method, process, and apparatus claims, in any patent *Licensable* by grantor.

Source Code shall mean the preferred form of the *Covered Software* for making modifications to it, including all modules it contains, plus any associated interface definition files, scripts used to control compilation and installation of an *Executable*, or source code differential comparisons against either the *Original Software* or another well known, available *Covered Software* of the *Contributor's* choice. The *Source Code* can be in a compressed or archival form, provided the appropriate decompression or de-archiving software is widely available for no charge.

You (or **Your**) shall mean an individual or a legal entity exercising rights under, and complying with all of the terms of, this *License* or a future version of this *License* issued under Section 6.1. For legal entities, **You** includes an entity which controls, is controlled by, or is under common control with *You*. For the purposes of this definition, **control** means (a) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (b) ownership of more than fifty per cent (50%) of the outstanding shares or beneficial ownership of such entity.

2. SOURCE CODE LICENSE

2.1 The Australian National University Grant.
Subject to the terms of this *License*, the *Australian National University* hereby grants *You* a world-wide, royalty-free, non-exclusive license, subject to third party intellectual property claims:
a) under copyrights *Licensable* by the *Australian National University* to use, reproduce, modify, display, perform, sublicense and distribute the *Original Software* (or portions thereof) with or without *Modifications*, and/or as part of a *Larger Work*;
b) and under *Patents Claims* infringed by the making, using or selling of *Original Software*, to make, have made, use, practice, sell, and offer for sale, and/or otherwise dispose of the *Original Software* (or portions thereof).
c) The licenses granted in this Section 2.1(a) and (b) are effective on the date the *Australian National University* first distributes *Original Software* under the terms of this *License*.
d) Notwithstanding Section 2.1(b) above, no patent license is granted:
1) for code that *You* delete from the *Original Software*;
2) separate from the *Original Software*; or
3) for infringements caused by: i) the modification of the *Original Software* or ii) the combination of the *Original Software* with other software or devices.

2.2 Contributor Grant.
Subject to the terms of this *License* and subject to third party intellectual property claims, each *Contributor* hereby grants *You* a world-wide, royalty-free, non-exclusive license:
a) under copyrights *Licensable* by *Contributor*, to use, reproduce, modify, display, perform, sublicense and distribute the *Modifications* created by such *Contributor* (or portions thereof) either on an unmodified basis, with other *Modifications*, as *Covered Software* and/or as part of a Larger Work; and
b) under *Patent Claims* necessarily infringed by the making, using, or selling of *Modifications* made by that *Contributor* either alone and/or in combination with its *Contributor Version* (or portions of such combination), to make, use, sell, offer for sale, have made, and/or otherwise dispose of:
1) *Modifications* made by that *Contributor* (or portions thereof); and
2) the combination of *Modifications* made by that *Contributor* with its *Contributor Version* (or portions of such combination).
c) The licenses granted in Sections 2.2(a) and 2.2(b) are effective on the date *Contributor* first makes *Commercial Use* of the *Covered Software*.
d) Notwithstanding Section 2.2(b) above, no patent license is granted:
1) for any code that *Contributor* has deleted from the *Contributor Version*;
2) separate from the *Contributor Version*; 3) for infringements caused by: i) third party modifications of *Contributor Version* or ii) the combination of *Modifications* made by that *Contributor* with other software (except as part of the *Contributor Version*) or other devices; or
4) under *Patent Claims* infringed by *Covered Software* in the absence of *Modifications* made by that *Contributor*.

3. DISTRIBUTION OBLIGATIONS

3.1 Application of License
The *Modifications* which *You* create or to which *You* contribute are governed by the terms of this *License*, including without limitation Section 2.2. The *Source Code* version of *Covered Software* may be distributed only under the terms of this *License* or a future version of this *License* released under Section 6.1, and *You* must include a copy of this *License* with every copy of the *Source Code You* distribute. *You* may not offer or impose any terms on any *Source Code* version that alters or restricts the applicable version of this *License* or the recipients' rights hereunder.

3.2 Availability of Source Code
Any *Modification* which *You* create or to which *You* contribute must be made available in *Source Code* form under the terms of this *License* either on the same media as an *Executable* version or via an accepted *Electronic Distribution Mechanism* to anyone to whom you made an *Executable* version available; and if made available via *Electronic Distribution Mechanism*, must remain available for at least twelve (12) months after the date it initially became available, or at least six (6) months after a subsequent version of that particular *Modification* has been made available to such recipients. You are responsible for ensuring that the *Source Code* version remains available even if the *Electronic Distribution Mechanism* is maintained by a third party.

3.3 Description of Modifications
You must cause all *Covered Software* to which You contribute to contain a file documenting the changes *You* made to create that *Covered Software* and the date of any change. *You* must include a prominent statement that the *Modification* is derived, directly or indirectly, from *Original Software* provided by the *Australian National University* and including the name of the *Australian National University* in (a) the *Source Code*, and (b) in any notice in an *Executable* version or related documentation in which *You* describe the origin or ownership of the *Covered Software*.

3.4 Intellectual Property Matters

a) Third Party Claims

If *Contributor* has knowledge that a license under a third party's intellectual property rights is required to exercise the rights granted by such *Contributor* under Sections 2.1 or 2.2, *Contributor* must include a text file with the *Source Code* distribution titled "LEGAL" which describes the claim and the party making the claim in sufficient detail that a recipient will know whom to contact. If *Contributor* obtains such knowledge after the *Modification* is made available as described in Section 3.2, *Contributor* shall promptly modify the LEGAL file in all copies *Contributor* makes available thereafter and shall take other steps (such as notifying appropriate mailing lists or newsgroups) reasonably calculated to inform those who received the *Covered Software* that new knowledge has been obtained.

b) Contributor APIs

If *Contributor's Modifications* include an application programming interface (API) and *Contributor* has knowledge of patent licenses which are reasonably necessary to implement that API, *Contributor* must also include this information in the LEGAL file.

c) Representations

Contributor represents that, except as disclosed pursuant to Section 3.4(a) above, *Contributor* believes that *Contributor's Modifications* are *Contributor's* original creation(s) and/or *Contributor* has sufficient rights to grant the rights conveyed by this *License*.

3.5 Required Notices

You must duplicate the notice in *Exhibit A* in each file of the *Source Code*. If it is not possible to put such notice in a particular *Source Code* file due to its structure, then *You* must include such notice in a location (such as a relevant directory) where a user would be likely to look for such a notice. If *You* created one or more *Modification(s)* *You* may add your name as a *Contributor* to the notice described in *Exhibit A*. You must also duplicate this *License* in any documentation for the *Source Code* where *You* describe recipients' rights or ownership rights relating to *Covered Software*. You may choose to offer, and to charge a fee for, warranty, support, indemnity or liability obligations to one or more recipients of *Covered Software*. However, *You* may do so only on *Your* own behalf, and not on behalf of the *Australian National University* or any *Contributor*. *You* must make it absolutely clear that any such warranty, support, indemnity or liability obligation is offered by *You* alone, and *You* hereby agree to indemnify the *Australian National University* and every *Contributor* for any liability incurred by the *Australian National University* or such *Contributor* as a result of warranty, support, indemnity or liability terms You offer.

3.6 Distribution of Executable Versions

You may distribute *Covered Software* in *Executable* form only if the requirements of Sections 3.1-3.5 have been met for that *Covered Software*, and if *You* include a notice stating that the *Source Code* version of the *Covered Software* is available under the terms of this *License*, including a description of how and where *You* have fulfilled the obligations of Section 3.2. The notice must be conspicuously included in any notice in an *Executable* version, related documentation or collateral in which *You* describe recipients' rights relating to the *Covered Software*. *You* may distribute the *Executable* version of *Covered Software* or ownership rights under a license of *Your* choice, which may contain terms different from this *License*, provided that *You* are in compliance with the terms of this *License* and that the license for the *Executable* version does not attempt to limit or alter the recipient's rights in the *Source Code* version from the rights set forth in this *License*. If *You* distribute the *Executable* version under a different license *You* must make it absolutely clear that any terms which differ from this *License* are offered by *You* alone, not by the *Australian National University* or any *Contributor*. *You* hereby agree to indemnify the *Australian National University* and every *Contributor* for any liability incurred by the *Australian National University* or such *Contributor* as a result of any such terms *You* offer.

3.7 Larger Works

You may create a *Larger Work* by combining *Covered Software* with other software not governed by the terms of this *License* and distribute the *Larger Work* as a single product. In such a case, *You* must make sure the requirements of this *License* are fulfilled for the *Covered Software*.

4. INABILITY TO COMPLY DUE TO STATUTE OR REGULATION

If it is impossible for *You* to comply with any of the terms of this *License* with respect to some or all of the *Covered Software* due to statute, judicial order, or regulation then *You* must: (a) comply with the terms of this *License* to the maximum extent possible; and (b) describe the limitations and the code they affect. Such description must be included in the LEGAL file described in Section 3.4 and must be included with all distributions of the *Source Code*. Except to the extent prohibited by statute or regulation, such description must be sufficiently detailed for a recipient of ordinary skill to be able to understand it.

5. APPLICATION OF THIS LICENSE

This *License* applies to code to which the *Australian National University* has attached the notice in *Exhibit A* and to related *Covered Software*.

6. VERSIONS OF THE LICENSE

6.1 New Versions

The *Australian National University* may publish revised and/or new versions of the *License* from time to time. Each version will be given a distinguishing version number.

6.2 Effect of New Versions

Once *Covered Software* has been published under a particular version of the *License*, *You* may always continue to use it under the terms of that version. *You* may also choose to use such *Covered Software* under the terms of any subsequent version of the *License* published by the *Australian National University*. No one other than the *Australian National University* has the right to modify the terms applicable to *Covered Software* created under this License.

7. DISCLAIMER OF WARRANTY
COVERED SOFTWARE IS PROVIDED UNDER THIS LICENSE ON AN "AS IS" BASIS, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, WARRANTIES THAT THE COVERED SOFTWARE IS FREE OF DEFECTS, MERCHANTABLE, FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGING. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE COVERED SOFTWARE IS WITH YOU. SHOULD ANY COVERED SOFTWARE PROVE DEFECTIVE IN ANY RESPECT, YOU (NOT THE AUSTRALIAN NATIONAL UNIVERSITY, ITS LICENSORS OR AFFILIATES OR ANY OTHER CONTRIBUTOR) ASSUME THE COST OF ANY NECESSARY SERVICING, REPAIR OR CORRECTION. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS LICENSE. NO USE OF ANY COVERED SOFTWARE IS AUTHORIZED HEREUNDER EXCEPT UNDER THIS DISCLAIMER.

8. TERMINATION

   8.1 This *License* and the rights granted hereunder will terminate automatically if *You* fail to comply with terms herein and fail to cure such breach within 30 days of becoming aware of the breach. All sublicenses to the *Covered Software* which are properly granted shall survive any termination of this *License*. Provisions which, by their nature, must remain in effect beyond the termination of this *License* shall survive.

   8.2 If *You* initiate litigation by asserting a patent infringement claim (excluding declatory judgment actions) against the *Australian National University* or a *Contributor* (the *Australian National University* or *Contributor* against whom You file such action is referred to as "Participant") alleging that:
   a) such Participant's *Contributor Version* directly or indirectly infringes any patent, then any and all rights granted by such *Participant* to *You* under Sections 2.1 and/or 2.2 of this *License* shall, upon 60 days notice from Participant terminate prospectively, unless if within 60 days after receipt of notice *You* either: (i) agree in writing to pay Participant a mutually agreeable reasonable royalty for *Your* past and future use of *Modifications* made by such Participant, or (ii) withdraw *Your* litigation claim with respect to the *Contributor Version* against such Participant. If within 60 days of notice, a reasonable royalty and payment arrangement are not mutually agreed upon in writing by the parties or the litigation claim is not withdrawn, the rights granted by Participant to *You* under Sections 2.1 and/or 2.2 automatically terminate at the expiration of the 60 day notice period specified above.
   b) any software, hardware, or device, other than such Participant's *Contributor Version*, directly or indirectly infringes any patent, then any rights granted to *You* by such Participant under Sections 2.1(b) and 2.2(b) are revoked effective as of the date *You* first made, used, sold, distributed, or had made, *Modifications* made by that Participant.

   8.3 If *You* assert a patent infringement claim against Participant alleging that such Participant's *Contributor Version* directly or indirectly infringes any patent where such claim is resolved (such as by license or settlement) prior to the initiation of patent infringement litigation, then the reasonable value of the licenses granted by such Participant under Sections 2.1 or 2.2 shall be taken into account in determining the amount or value of any payment or license.

   8.4 In the event of termination under Sections 8.1 or 8.2 above, all end user license agreements (excluding distributors and resellers) which have been validly granted by *You* or any distributor hereunder prior to termination shall survive termination.

9. LIMITATION OF LIABILITY
UNDER NO CIRCUMSTANCES AND UNDER NO LEGAL THEORY, WHETHER TORT (INCLUDING NEGLIGENCE), CONTRACT, OR OTHERWISE, SHALL YOU, THE AUSTRALIAN NATIONAL UNIVERSITY, ANY OTHER CONTRIBUTOR, OR ANY DISTRIBUTOR OF COVERED SOFTWARE, OR ANY SUPPLIER OF ANY OF SUCH PARTIES, BE LIABLE TO ANY PERSON FOR ANY INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY CHARACTER INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF GOODWILL, WORK STOPPAGE, COMPUTER FAILURE OR MALFUNCTION, OR ANY AND ALL OTHER COMMERCIAL DAMAGES OR LOSSES, EVEN IF SUCH PARTY SHALL HAVE BEEN INFORMED OF THE POSSIBILITY OF SUCH DAMAGES. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO LIABILITY FOR DEATH OR PERSONAL INJURY RESULTING FROM SUCH PARTY'S NEGLIGENCE TO THE EXTENT APPLICABLE LAW PROHIBITS SUCH LIMITATION. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, BUT MAY ALLOW LIABILITY TO BE LIMITED; IN SUCH CASES, A PARTY'S, ITS EMPLOYEES', LICENSORS' OR AFFILIATES' LIABILITY SHALL BE LIMITED TO AUD $100. NOTHING CONTAINED IN THIS LICENSE SHALL PREJUDICE THE STATUTORY RIGHTS OF ANY PARTY DEALING AS A CONSUMER.

10. MISCELLANEOUS. This *License* represents the complete agreement concerning subject matter hereof. All rights in the *Covered Software* not expressly granted under this *License* are reserved. Nothing in this *License* shall grant *You* any rights to use any of the trademarks of the *Australian National University* or any of its Affiliates, even if any of such trademarks are included in any part of *Covered Software* and/or documentation to it. This *License* is governed by the laws of the Australian Capital Territory excluding its conflict-of-law provisions. All disputes or litigation arising from or relating to this Agreement shall be subject to the jurisdiction of the Supreme Court of the Australian Capital Territory. If any part of this Agreement is found void and unenforceable, it will not affect the validity of the balance of the Agreement, which shall remain valid and enforceable according to its terms.

11. RESPONSIBILITY FOR CLAIMS
As between the *Australian National University* and the *Contributors*, each party is responsible for claims and damages arising, directly or indirectly, out of its utilisation of rights under this *License* and *You* agree to work with the *Australian National University* and *Contributors* to distribute such responsibility on an equitable basis. Nothing herein is intended or shall be deemed to constitute any admission of liability.

EXHIBIT A

The contents of this file are subject to the ANUOS License Version 1.0 (the "License"); you may not use this file except in compliance with the License. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. The Original Software is _____. The Initial Developers of the Original Software are Dr Peter Christen (Department of Computer Science, Australian National University), Dr Tim Churches (Centre for Epidemiology and Research, New South Wales Department of Health) and Drs Markus Hegland, Stephen Roberts and Ole Nielsen (Mathematical Sciences Insitute, Australian National University). Copyright (C) 2002 the Australian National University and others. All Rights Reserved. Contributors: _____

APPENDIX 1

DIFFERENCES BETWEEN THE ANUOS LICENSE VERSION 1.0, THE MOZILLA PUBLIC LICENSE VERSION 1.1 AND THE NOKIA OPEN SOURCE LICENSE (NOKOS LICENSE) VERSION 1.0A

The ANUOS License Version 1.0 was derived from the Mozilla Public License Version 1.1 using some of the changes to the Mozilla Public License embodied in the Nokia Open Source License (NOKOS License) Version 1.0a. The differences between the ANUOS License Version 1.0 (this document), the Mozilla Public License and the NOKOS License are as follows:

i. The title of the license was changed to "Australian National University Open Source License (ANUOS License) Version 1.0".

ii. Globally, all references to "Netscape Communications Corporation", "Mozilla", "Nokia" and "Nokia Corporation" were changed to "Australian National University".

iii. Globally, the words "means", "Covered Code" and "Covered Software" as used in the Mozilla Public License were changed to "shall mean", "Covered Code" and "Covered Software" respectively, as used in the NOKOS License.

iv. In Section 1 (Definitions) and Exhibit A, a definition of "the Australian National University" was added, a definition of "Associated Documentation and Data Files" was added and the definitions of "Covered Software", "Original Software" and "Modifications" were expanded to include "Associated Documentation and Data Files".

v. In Section 2, the term "intellectual property rights" used in the Mozilla Public License was replaced by the term "copyrights" as used in the NOKOS License.

vi. In Section 2.2 (Contributor Grant), the words "Subject to the terms of this License" which appear in the NOKOS License were added to the Mozilla Public License.

vii. The sentence "However, You may include an additional document offering the additional rights described in Section 3.5." which appears in the Mozilla Public License was omitted.

viii. Section 6.3 (Derivative Works) of the Mozilla Public License, which permits modifications to the Mozilla Public License, was omitted.

ix. In Section 9 (Limitation of Liability), a maximum liability of AUD $100 was specified for those jurisdictions which do not allow complete exclusion of liability but which do allow limitation of liability. The sentence "NOTHING CONTAINED IN THE LICENSE SHALL PREJUDICE THE STATUTORY RIGHTS OF ANY PARTY DEALING AS A CONSUMER.", which appears in the NOKOS License but not in the Mozilla Public License, was added.

x. Section 10 of the Mozilla Public License, which provides additional conditions for United States Government End Users, was omitted.

xi. The governing law and jurisdiction for the settlement of disputes in Section 11 of the Mozilla Public License and Section 10 of the NOKOS License was changed to the laws of the Australian Capital Territory and the Supreme Court of the Australian Capital Territory respectively. The exclusion of the application of the United Nations Convention on Contracts for the International Sale of Goods which appears in the Mozilla Public License was omitted.

xii. Section 13 (Multiple-Licensed Code) of the Mozilla Public License was omitted.

xiii. The provisions for alternative licensing arrangement for contributed code which appear in Exhibit A of the Mozilla Public License were omitted.

xiv. In Exhibit A the names of the Australian National University staff members who developed the software are specified in the identification of the Initial Developer.

BIBLIOGRAPHY

[1] G.B. Bell and A. Sethi, *Matching Records in a National Medical Patient Index*, Communications of the ACM, Vol. 44 No. 9, September 2001.

[2] V. Borkar, K. Deshmukh and S. Sarawagi, *Automatic segmentation of text into structured records*, in Proceedings of the 2001 ACM SIGMOD international conference on Management of Data, Santa Barbara, California, 2001.

[3] W.W. Cohen, *The WHIRL Approach to Integration: An Overview*, in Proceedings of the AAAI-98 Workshop on AI and Information Integration. AAAI Press, 1998.

[4] I. Fellegi and A. Sunter, *A Theory for Record Linkage*. In Journal of the American Statistical Society, 1969.

[5] H. Galhardas, D. Florescu, D. Shasha and E. Simon, *An Extensible Framework for Data Cleaning*, Technical Report 3742, INRIA, 1999.

[6] L. Gill, *Methods for Automatic Record Matching and Linking and their use in National Statistics*, National Statistics Methodology Series No. 25, London 2001.

[7] J. Han and M. Kamber, *Data Mining: Concepts and Techniques*, Morgan Kaufmann, 2000.

[8] M.A. Hernandez and S.J. Stolfo, *The Merge/Purge Problem for Large Databases*, in Proceedings of the SIGMOD Conference, San Jose, 1995.

[9] C.W. Kelman, *Monitoring Health Care Using National Administrative Data Collections*, PhD thesis, Australian National University, Canberra, May 2000.

[10] A.J. Lait, and B. Randell, *An Assessment of Name Matching Algorithms*, Technical Report, Department of Computing Science, University of Newcastle upon Tyne, UK 1993.

[11] J.I. Maletic and A. Marcus, *Data Cleansing: Beyond Integrity Analysis*, in Proceedings of the Conference on Information Quality (IQ2000), Boston, October 2000.

[12] *AutoStan and AutoMatch, User's Manuals*, MatchWare Technologies, Kennebunk, Maine, 1998. See also: www.fcsm.gov/working-papers/software-demos.pdf

[13] H.B. Newcombe and J.M. Kennedy, *Record Linkage: Making Maximum Use of the Discriminating Power of Identifying Information*, Communications of the ACM, Vol. 5 No. 11, 1962.

[14] L. Philips, *The Double-Metaphone Search Algorithm*, C/C++ User's Journal, Vol. 18 No. 6, June 2000.

[15] E.H. Porter and W.E. Winkler, *Approximate String Comparison and its Effect on an Advanced Record Linkage System*, Research Report RR97/02, US Bureau of the Census, 1997.

[16] L.R. Rabiner, *A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition*, in Proceedings of the IEEE, vol. 77, no. 2, February 1989.

[17] E. Rahm and H.H. Do, *Data Cleaning: Problems and Current Approaches*, IEEE Bulletin of the Technical Committee on Data Engineering, Vol. 23 No. 4, December 2000.

[18] K. Seymore. A. McCallum and R. Rosenfeld, *Learning Hidden Markov Model Structure for Information Extraction*, in Proceedings of AAAI-99 Workshop on Machine Learning for Information Extraction, 1999.

[19] V.S. Verykios, A.K. Elmagarmid and E.N. Houstis, *Automating the Approximate Record-Matching Process*, Information Sciences, Vol. 126, July 2000.

[20] W.E. Winkler, *Quality of Very Large Databases*, Research Report RR2001/04, US Bureau of the Census, 2001.

[21] W.E. Yancey, *Frequency-Dependent Probability Measures for Record Linkage*, Research Report RR00/07, Statistical Research Division, US Bureau of the Census, July 2000.

[22] W.E. Yancey, *BigMatch: A Program for Extracting Probable Matches from a Large File for Record Linkage*, Research Report RR 2000-01, Statistical Research Division, US Bureau of the Census, March 2002.

INDEX