Febru – Freely extensible biomedical record linkage

Release 0.2.2

Peter Christen [1], Tim Churches [2]

November 13, 2003

[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
Copyright © 2002, 2003 Australian National University. All rights reserved.

See Appendix H of this document for the license conditions under which this document and the computer programs described in it may be used.
Abstract

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

This fourth release **Febrl** Version 0.2.2 is a bug-fix release of Version 0.2.1. The main features of the current release are:

- Probabilistic and rules-based cleaning and standardisation routines for names, addresses and dates.
- A variety of supplied look-up and frequency tables for names and addresses.
- Various comparison functions for names, addresses, dates and localities, including approximate string comparisons, phonetic encodings, geographical distance comparisons, and time and age comparisons.
- Several blocking (indexing) methods, including the traditional compound key blocking used in many record linkage programs.
- Probabilistic record linkage routines based on the classical Fellegi and Sunter approach, as well as a *flexible classifier* that allows a flexible definition of the weight calculation.
- Process indicators that give estimations of remaining processing times.
- Access methods for fixed format and comma-separated value (CSV) text files, as well as SQL databases.
- Efficient temporary direct random access data set based on the *Berkeley database library*.
- One-to-one assignment procedure for linked record pairs based on the *Auction* algorithm.
- Supports parallelism for higher performance on parallel platforms, based on **MPI** (Message Passing Interface), a standard for parallel programming, and **Pypar**, an efficient and easy-to-use module that allows Python programs to run in parallel on multiple processors and communicate using MPI.
- A database generator which allows the creation of data sets of randomly created records (containing names, addresses and dates) with the possibility to include duplicate records with randomly introduced modifications. This allows for easy testing and evaluation of linkage (deduplication) processes.
- Example project modules and example data sets allowing simple running of **Febrl** projects without any modifications needed.
- This extensive manual.

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**.

See Also:

- **Febrl Project Web Site**
  (http://datamining.anu.edu.au/linkage.html) for information about this project.
- **Python Web Site**
  (http://www.python.org/) for information on the Python programming language.
• **MPI (Message Passing Interface) Web Site**
  (http://www-unix.mcs.anl.gov/mpi/)
  for information on MPI.

• **Pypar Web Site**
  (http://datamining.anu.edu.au/pypar)
  for information on Pypar.

• **MySQL Web Site**
  (http://www.mysql.com)
  for information on the open source database MySQL.

• **MySQL for Python**
  (http://sourceforge.net/projects/mysql-python)
  for information on the Python MySQL module.

• **Sleepycat Software**
  (http://www.sleepycat.com/)
  for information on the Berkeley database library.

• **Python Bindings for BerkeleyDB**
  (http://pybsddb.sourceforge.net/)
  for information on the Python bsddb3 module.

• **Logilab HMM Python module**
  (http://www.logilab.org/projects/hmm/)
  for information on Logilab’s Python hidden Markov model hmm module.

This document is subject to the ANUOS License Version 1.1 (the License, see Appendix H of this document); you may not use this document except in compliance with the License. All Fehrl computer program code and associated data files and documentation, including this document, are distributed under the License 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.
# CONTENTS

1 Acknowledgments .................................................. 1

2 Introduction ...................................................... 3
   2.1 Performance ............................................... 5

3 Data Cleaning and Record Linkage ............................. 9

4 System Overview .................................................. 11

5 Configuration and Running Febrl using a Module derived from 'project.py' 13

6 Data Cleaning and Standardisation ............................. 27
   6.1 Name and Address Cleaning and Standardisation ............. 28
   6.2 Output Fields ............................................. 30
   6.3 Name Cleaning and Standardisation using a Rules Based Approach .................................. 30
   6.4 Name Cleaning and Standardisation using a Hidden Markov Model Based Approach ............. 32
   6.5 Address Cleaning and Standardisation using a Hidden Markov Model Based Approach ............. 33
   6.6 Date Cleaning and Standardisation .......................... 35
   6.7 Field Passing .............................................. 37
   6.8 Record Cleaning and Standardisation ........................ 38
   6.9 Starting a Standardisation Process .......................... 39

7 Hidden Markov Models for Data Standardisation .......... 41
   7.1 Hidden Markov Model Implementation Module 'simplehmm.py' .................................. 43

8 Hidden Markov Model Training .................................. 47
   8.1 Program 'tagdata.py' .................................... 49
   8.2 Program 'trainhmm.py' .................................... 53

9 Record Linkage and Deduplication ............................. 55
   9.1 Indexing .................................................. 55
   9.2 Field Comparison Functions ................................ 59
   9.3 Record Comparator ........................................ 66
   9.4 Example Field and Record Comparator Initialisation ........ 67
   9.5 Classification ............................................ 68
   9.6 Starting a Linkage or Deduplication Process .............. 71

10 Output .......................................................... 73
   10.1 Record Pair One-To-One Assignment Restrictions ........ 76
Acknowledgments

This project is funded by the Australian National University (ANU) and the NSW Department of Health under an AICS (ANU-Industry Collaboration Scheme) AICS #1-2001. Additional funding is provided by the Australian Partnership for Advanced Computing (APAC). 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 Xi Zhu (former honours student at the ANU Department of Computer Science) for his recognition of the potential of hidden Markov models in data standardisation and his documentation of the rule based name standardisation (Appendix C), 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.

We would also like to thank Peter Viechnicki (Vredenburg Corp., Lanham, MD), Rohan Baxter (CSIRO, Canberra) and Daniel R. Sabath (Harborview Injury Prevention and Research Center, Seattle) for their feedback and contributions, and all other users of **Febrl** who sent us bug-reports or other comments.
Introduction

Record linkage is a rapidly growing field with applications in many areas of health and biomedical research [1, 10, 13]. 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 [11]. Often the data required for such analyses are contained in two or more separate databases, which do not share a common unique entity 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 [8] in 1969, and newer approaches using maximum entropy and other machine learning techniques [6, 7, 17, 25, 26].

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 otherwise not 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 **Febrl** (‘**Freely extensible biomedical record linkage**’), 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 H), 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 E.

Since its initial release (Version 0.1) the **Febrl** system has undergone a major redesign resulting in an object-orient approach which allows easier configuration and is more extensible. This fourth release (Version 0.2.1) is a bug-fix release for **Febrl** Version 0.2.1, and contains routines not only to clean and standardise data sets, but also to link and deduplicate them.

It is assumed that the reader has at least superficial familiarity with the syntax of the Python programming language in order to understand and customise the main 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](http://www.python.org). 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. Some ideas on the performance of **Febrl** (i.e. how long it takes to standardise and link or deduplicate certain numbers of records) is given in Section 2.1. 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 description of the central `project.py` module which needs to be modified by a user to control the **Febrl** system and to run data cleaning and standardisation, as well as deduplication and linkage processes. The task of data cleaning and standardisation as implemented in **Febrl** is then presented in more detail in Chapter 6, including how to define and run a standardisation process. Name and address standardisation in **Febrl** is done using hidden Markov models (HMMs), and this technique is introduced in Chapter 7. HMMs are a powerful alternative to the often cumbersome rules-based approach to data standardisation. Chapter 8 then deals with the issue of HMM training. Instructions for the use of the programs `tagdata.py` and `trainhmm.py` are given in this chapter. Descriptions of the various components of the record linkage and deduplication processes are given in Chapter 9, including how to define field comparison functions, indexing techniques and matching classifiers. Several output forms are supported by **Febrl**, including a histogram, printing of record pairs, as well as saving results into text files. Chapter 10 presents these output forms in more detail. Also discussed in Chapter 10 are assignment restrictions, which can for example be applied to force one-to-one assignments for record pairs. Currently two auxiliary programs (`randomselect.py` which allows random selection of input records and `generate.py`, which is a database generator able to create records and duplicates) are provided and described in Chapter 11. Access to various data set formats is provided in **Febrl** and this is the topic of Chapter 12. The various look-up tables and their corresponding
file formats are described in Chapter 13. The **Febrl** system is provided with logging and verbose output capabilities, and Chapter 14 shows how to define and configure a *project logger*. Finally, the installation of the **Febrl** system is discussed in Chapter 15, and information on how to run **Febrl** on a parallel platform using MPI and Pypar is given in Chapter 16. Note that parallelism within **Febrl** is in its initial stage, and we would like to ask people who are interested in this area to contact the authors for further exchange of detailed information and experiences.

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. A description of the rule-based name standardisation as implemented in **Febrl** is given in Appendix C. The manifest in Appendix D 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 E. All files provided with the current **Febrl** version are listed in Appendix F, and in Appendix G support arrangements are discussed. Finally, a copy of the ANU Open Source License can be found in Appendix H.

---

**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](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](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.

---

### 2.1 Performance

To give an idea on the performance of **Febrl** we present timing results of experiments we made on our computing platform, a *SUN Enterprise 450* shared memory (SMP) server with four 480 MHz *Ultra-SPARC II* processors and 4 Giga Bytes of main memory.

We were running deduplication processes with 20,000, 100, 000 and 200, 000 records from a health data set containing midwife data records provided by the *NSW Department of Health*. This data set has earlier been standardised into a clean form and is stored as a CSV (comma separated values) text file. Six field comparison functions were used and the classical blocking index technique with three indexes (passes) was applied. The standard *Fellegi and Sunter* classifier with a lower threshold value of 0.0 and an upper threshold value of 30.0 was used to classify the record pairs, and finally a one-to-one assignment procedure was applied.

We ran these deduplication tasks using a varying number of processors (1, 2, 3 and 4) and the achieved results are shown in tables 2.1, 2.2, 2.3 and 2.4 in the form *hh:mm:ss* or *mm:ss*, respectively. Step 1 denotes the loading for records and building of the indexes, step 2 the actual deduplication and step 3 the one-to-one assignment and saving the result into text files. Time spent for communication is given in seconds.

A closer look at the results shows that parallel processing of **Febrl** results in almost linear speedup, i.e. running **Febrl** on two processors means the run time is reduced by a factor of two, and on four processors the same task is around 3.7 times faster than on one processor. The results also show that the dominating factor is the comparison of record pairs, which unfortunately is not linearly scalable with the number of records. The increase in records from 20,000 to 200,000 (10-fold) results in a record pair comparison time increased by a factor of around thirty.
Table 2.1: Deduplication performance of 20,000 records using a memory based temporary data set.

<table>
<thead>
<tr>
<th>Times</th>
<th>Sequential</th>
<th>2 processors</th>
<th>3 processors</th>
<th>4 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>2 : 18 : 42</td>
<td>1 : 09 : 06</td>
<td>48 : 35</td>
<td>37 : 38</td>
</tr>
<tr>
<td>Step 1</td>
<td>4 : 57</td>
<td>2 : 57</td>
<td>2 : 17</td>
<td>1 : 44</td>
</tr>
<tr>
<td>Step 2</td>
<td>2 : 12 : 58</td>
<td>1 : 05 : 20</td>
<td>45 : 25</td>
<td>35 : 01</td>
</tr>
<tr>
<td>Step 3</td>
<td>0 : 43</td>
<td>0 : 40</td>
<td>0 : 39</td>
<td>0 : 37</td>
</tr>
<tr>
<td>Communication in step 1</td>
<td>–</td>
<td>3.7 sec</td>
<td>5.2 sec</td>
<td>7.3 sec</td>
</tr>
<tr>
<td>Communication in step 2</td>
<td>–</td>
<td>3.2 sec</td>
<td>2.8 sec</td>
<td>5.6 sec</td>
</tr>
</tbody>
</table>

Table 2.2: Deduplication performance of 20,000 records using a disk based temporary data set.

<table>
<thead>
<tr>
<th>Times</th>
<th>Sequential</th>
<th>2 processors</th>
<th>3 processors</th>
<th>4 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>3 : 07 : 44</td>
<td>1 : 33 : 45</td>
<td>1 : 07 : 18</td>
<td>50 : 06</td>
</tr>
<tr>
<td>Step 1</td>
<td>5 : 15</td>
<td>3 : 14</td>
<td>2 : 26</td>
<td>1 : 58</td>
</tr>
<tr>
<td>Step 2</td>
<td>3 : 01 : 43</td>
<td>1 : 29 : 46</td>
<td>1 : 04 : 08</td>
<td>47 : 25</td>
</tr>
<tr>
<td>Step 3</td>
<td>0 : 42</td>
<td>0 : 38</td>
<td>0 : 38</td>
<td>0 : 37</td>
</tr>
<tr>
<td>Communication in step 1</td>
<td>–</td>
<td>3.2 sec</td>
<td>4.2 sec</td>
<td>5.3 sec</td>
</tr>
<tr>
<td>Communication in step 2</td>
<td>–</td>
<td>3.4 sec</td>
<td>3.0 sec</td>
<td>4.2 sec</td>
</tr>
</tbody>
</table>

Table 2.3: Deduplication performance of 100,000 records using a disk based temporary data set.

<table>
<thead>
<tr>
<th>Times</th>
<th>Sequential</th>
<th>2 processors</th>
<th>3 processors</th>
<th>4 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>50 : 08 : 34</td>
<td>15 : 22 : 06</td>
<td>10 : 20 : 11</td>
<td>8 : 07 : 22</td>
</tr>
<tr>
<td>Step 1</td>
<td>26 : 21</td>
<td>16 : 45</td>
<td>12 : 52</td>
<td>10 : 10</td>
</tr>
<tr>
<td>Step 3</td>
<td>10 : 57</td>
<td>9 : 33</td>
<td>8 : 43</td>
<td>8 : 23</td>
</tr>
<tr>
<td>Communication in step 1</td>
<td>–</td>
<td>14 sec</td>
<td>20 sec</td>
<td>25 sec</td>
</tr>
<tr>
<td>Communication in step 2</td>
<td>–</td>
<td>19 sec</td>
<td>30 sec</td>
<td>50 sec</td>
</tr>
</tbody>
</table>

Table 2.4: Deduplication performance of 200,000 records using a disk based temporary data set.

<table>
<thead>
<tr>
<th>Times</th>
<th>Sequential</th>
<th>2 processors</th>
<th>3 processors</th>
<th>4 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>53 : 22</td>
<td>35 : 00</td>
<td>26 : 18</td>
<td>21 : 25</td>
</tr>
<tr>
<td>Step 2</td>
<td>104 : 25 : 32</td>
<td>51 : 52 : 37</td>
<td>35 : 00 : 37</td>
<td>27 : 41 : 55</td>
</tr>
<tr>
<td>Step 3</td>
<td>1 : 00 : 16</td>
<td>59 : 09</td>
<td>52 : 45</td>
<td>54 : 53</td>
</tr>
<tr>
<td>Communication in step 1</td>
<td>–</td>
<td>28 sec</td>
<td>45 sec</td>
<td>53 sec</td>
</tr>
<tr>
<td>Communication in step 2</td>
<td>–</td>
<td>132 sec</td>
<td>226 sec</td>
<td>284 sec</td>
</tr>
</tbody>
</table>

Table 2.5 shows the maximal amount of memory used by the various Febrl test runs. Similarly to the increased run times, the amount of memory needed increases more than linearly. A ten-fold increase in the number of records results in a fifteen-fold increase in the amount of memory needed. Additionally, parallel processing in Febrl also results in an increased amount of memory needed, which is due to the replication of various data structures on the parallel Febrl processes.

These timing and memory results given here are just examples on the performance of Febrl on a particular platform given a particular data set and performing deduplication as defined in an example project file. Potential users should
Table 2.5: Maximal amount of memory used by deduplication process (in Mega Bytes).

<table>
<thead>
<tr>
<th>Deduplication task</th>
<th>Sequential</th>
<th>2 processors</th>
<th>3 processors</th>
<th>4 processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>20,000 records, memory data set</td>
<td>89</td>
<td>192</td>
<td>276</td>
<td>360</td>
</tr>
<tr>
<td>20,000 records, disk data set</td>
<td>60</td>
<td>134</td>
<td>188</td>
<td>242</td>
</tr>
<tr>
<td>100,000 records, disk data set</td>
<td>238</td>
<td>603</td>
<td>820</td>
<td>1,005</td>
</tr>
<tr>
<td>200,000 records, disk data set</td>
<td>906</td>
<td>2,130</td>
<td>2,829</td>
<td>3,495</td>
</tr>
</tbody>
</table>

Note that many factors influence the performance of Febrl, including the definition of the standardisation, blocking indexes and linkage or deduplication processes, as well as the given computing platform and data sets.
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 entity 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 [12] 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 [19] in 1962 while the theoretical foundation was provided by Fellegi and Sunter [8] in 1969. Using frequency counts [29] to derive agreement and disagreement probabilities, each pair of fields of each compared record pair 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 [14] 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 [21] 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 [25] 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 [28] to be able to handle such large volumes of data. Sorting large number of records becomes the main bottleneck, so extracting sub-sets of possible links from an unsorted large data file [30] has to be done as a pre-processing step before the actual record comparisons can be done. The authors of [7, 26] describe a hybrid system that in a first step uses unsupervised clustering on a small sample data set to create data that can be used by a classifier in the second step to classify records into links and non-links. The authors do not directly address the problem of data cleaning and standardisation, rather they use approximate string comparison algorithms to be able to deal with variations in strings. Clustering of large and high-dimensional data sets with applications to matching is discussed in [17]. The authors propose a two step clustering algorithm that in the first step uses cheap approximate distance metrics to form overlapping canopies, which are in a second step clustered using traditional approaches. Another approach [18] learns field specific string-edit distance weights and a binary classifier based on support vector machines (SVM) to find duplicate records in text databases.

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 [23]. 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 [6] between such vectors. Another possibility is to use an SQL like language [9] 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 [15] 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 [12] 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, many don’t consider the statistical theory of record linkage as developed by Fellegi and Sunter [8] 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.
The **Febrl** system is implemented in an object oriented design with a handful of modules, each containing routines for specific tasks. The overall system is configured and controlled by a `project.py` module, which will be explained in detail in Chapter 5.

Record linkage consists of two main steps. The first one deals with data cleaning and standardisation, while the second performs the actual linkage (or deduplication). The user thus needs to specify various settings in order to be able to perform a cleaning/standardisation and/or a linkage/deduplication process.

- **Definition of data sets**
  The two different types of data sets needed by **Febrl** are input data sets (i.e. the raw and maybe uncleaned original data) and temporary data sets (to hold cleaned and standardised records before they are linked or used in deduplication). Data sets are described in detail in Chapter 12.

- **Definition of look-up tables**
  Various look-up tables are needed in the cleaning (correction lists), standardisation (tagging look-up tables) and the linkage and deduplication (frequency and geocode look-up tables) processes. Chapter 13 describes the various types of look-up tables available, their file formats and shows how to initialise and load them.

- **Definition of hidden Markov models (HMMs)**
  For name and address segmentation within the data standardisation process HMMs can be used efficiently instead of a traditional rules based approach. The application of HMMs in data standardisation in general is described in Chapter 7, while Chapter 8 explains the training process for HMMs as used within **Febrl** in more detail.

- **Definitions for date standardisation**
  For the standardisation of dates, special data format strings need to be given, as described in Section 6.6.

- **Definition of standardisation processes**
  This includes which fields from input data sets should be cleaned and standardised using what methods and how should the resulting cleaned and standardised records be written into temporary or output data sets. Chapter 6 deals in detail with these issues.

- **Definition of blocking indexes**
  To reduce the huge number of possible comparisons between record pairs, indexing methods (like blocking or sorting) are used. These methods are described in Section 9.1.

- **Definition of field comparison functions**
  In the record linkage and deduplication processes, record pairs are compared field by field using one of the available field comparison functions which are described in Section 9.2.

- **Definition of a classifier**
  The weight vectors resulting from record comparisons need to be classified into either links, non-links and possible links, using one of the available classifiers. Section 9.5 describes these classifiers.
• Definition of logging and verbose output
  Finally, the Febrl system is capable to write log information into a file and display it to standard output at various levels of volume. Furthermore, warning and error messages that are generated by the system can be logged and displayed as well. The setting up of a project logger (which should be done at the beginning of a project.py module) is described in detail in Chapter 14.

• Definition of output forms
  Several output forms are possible with the current version of Febrl, including displaying a histogram and detailed views of record pairs, as well as saving record pairs into text files. Chapter 10 deals with these issues.

• Setting of output assignment restrictions
  After the classification of record pairs Febrl allows the application of a one-to-one assignment procedure which finds optimal assignments of record pairs. Section 10.1 deals with this topics in more details.

• Starting a standardisation, linkage or deduplication process
  Once all necessary settings are done, it is very simple to define and start a standardisation, linkage or deduplication process, as shown in Section 9.6.
All aspects of the Febrl system are configured and controlled by a single Python module (program) that can be derived from either the `project-standardise.py`, `project-linkage.py` or `project-deduplicate.py` modules provided. In this section the complete `project-deduplicate.py` module as supplied with the current Febrl distribution is described in detail using blocks of code extracted from this module. The module `project-linkage.py` is very similar in its structure, with the main difference that two data sets are dealt with, while the module `project-standardise.py` only contains the definitions for one input and one output data set plus the necessary standardisers, but no linkage or deduplication processes are defined. Each code block is explained and references to the relevant chapters are given. It is assumed that the reader has some familiarity with the (very simple) syntax of the Python programming language in which Febrl is implemented. If not, the necessary knowledge can be gained in just a few hours from one of the tutorials or introductions listed on the Python Web site at http://www.python.org. Note that comments in the Python language start with a hash character (#) and continue until the end of a line. Alternatively a comment can go over several lines if it is enclosed by triple quotes """ as shown in the code below for the module name documentation string.

At the top of the `project-deduplicate.py` module is the header with version and licensing information, followed by a documentation string that gives the name of the module and a short description.
Briefly, what needs to be defined for a deduplication project is:
- A Febrl object, a project, plus a project logger
- One input data set
- One corresponding temporary data set (with ‘readwrite’ access)
- Lookup tables to be used
- Standardisers for names, addresses and dates
- Field comparator functions and a record comparator
- A blocking index
- A classifier

and then the ‘deduplicate’ method can be called.

This project module will standardised and then deduplicate the example data set ‘dataset2.csv’ given in the ‘dbgen’ directory.
In the following code block all the required Febrl modules are imported so the necessary functionality is available.

```python
# Imports go here
import sys # Python system modules needed
import time
from febrl import * # Main Febrl classes
from dataset import * # Data set routines
from standardisation import * # Standardisation routines
from comparison import * # Comparison functions
from lookup import * # Look-up table routines
from indexing import * # Indexing and blocking routines
from simplehmm import * # Hidden Markov model (HMM) routines
from classification import * # Classifiers for weight vectors
```

Next the system is initialised by creating a Febrl object `myfebrl`, and a new project is initialised as part of it. This project is assigned a name, a description and a file name (into which it can be saved at any time). Febrl loads records block wise from the input data sets, and the argument `block_size` controls how many records should be loaded into one such block.

Note that you can use names other than `myfebrl` and `myproject` if you wish – just be sure to use the same name throughout. In fact, you can configure multiple Febrl projects from one `project.py` module if you wish, using different variable names to refer to each project object. For the sake of simplicity, we will only configure one project here.

```python
# Set up Febrl and create a new project (or load a saved project)
myfebrl = Febrl(description = 'Example Febrl instance',
                 febrl_path = '.
myproject = myfebrl.new_project(name = 'example-dedup',
                                 description = 'Deduplicate example data set',
                                 file_name = 'example-deduplicate.fbr',
                                 block_size = 1000,
                                 parallel_write = 'host')
```

The argument `block_size` sets the number of records that will be loaded and processed in a block because Febrl is working in a blocking fashion. Note that this block size has nothing to do with the blocking as used within the record linkage process. Rather, records are loaded from files in chunks (blocks) and processed, before the next block is loaded and processed.

The `parallel_write` argument sets the way in which data sets are written to files when Febrl is run in parallel. If set to ‘host’ only the host process (the one Febrl has been started on) writes into data sets. On the other hand, if set to ‘all’ then all processes write into local files. See Chapter 12 for more detailed information on this topic.

To enable verbose output and logging of status, warning and error messages into a log file, a project logger is defined next. The verbose and log output levels can be set independently from each other to values 0 (no output), 1 (only summary output), 2 (extended summary output) or 3 (very detailed output at record level). Similar to the `parallel_write` argument discussed above, `parallel_print` (which can be set to ‘host’ or ‘all’) defines the way output is logged and printed. For more details on verbose output and logging please see Chapter 14.

---

15
# Define a project logger

mylog = ProjectLog(file_name = 'example-dedup.log',
                   project = myproject,
                   log_level = 1,
                   verbose_level = 2,
                   clear_log = True,
                   no_warn = False,
                   parallel_print = 'host')

The next two code blocks define an input data set (initialised for reading from a CSV file) and a temporary memory based data set (to hold the cleaned and standardised records before they are deduplicated or linked and written into an output data set). See Chapter 12 for more information on the given arguments, and how to access data sets. Note that this initialisation of data sets does not mean that they are immediately loaded, but only that preparations are made to access them. Note also that fields defined in the input data set do not need to be all the columns in the data file or database table being accessed (if for example columns are not needed in a standardisation and linkage process).

**Note:** While in this example the temporary data set is a memory based data set, for larger projects (with data sets containing more than a couple of thousand records) it is recommended that a *Shelve* data set (file based) is used as temporary data set (see Chapter 12 for more information on this topic).

# Define your original input data set(s)

# Only one data set is needed for deduplication

indata = DataSetCSV(name = 'example2in',
                    description = 'Example data set 2',
                    access_mode = 'read',
                    header_lines = 1,
                    file_name = './dbgen/dataset2.csv',
                    fields = {'rec_id':0,
                              'given_name':1,
                              'surname':2,
                              'street_num':3,
                              'address_part_1':4,
                              'address_part_2':5,
                              'suburb':6,
                              'postcode':7,
                              'state':8,
                              'date_of_birth':9,
                              'soc_sec_id':10},
                    fields_default = '',
                    strip_fields = True,
                    missing_values = ['','missing'])
# Define a temporary data set

tmpdata = DataSetMemory(name = 'example2tmp',
                        description = 'Temporary example 2 data set',
                        access_mode = 'readwrite',
                        fields = {'title':1,
                                  'gender_guess':2,
                                  'given_name':3,
                                  'alt_given_name':4,
                                  'surname':5,
                                  'alt.surname':6,
                                  'wayfare_number':7,
                                  'wayfare_name':8,
                                  'wayfare_qualifier':9,
                                  'wayfare_type':10,
                                  'unit_number':11,
                                  'unit_type':12,
                                  'property_name':13,
                                  'institution_name':14,
                                  'institution_type':15,
                                  'postaddress_number':16,
                                  'postaddress_type':17,
                                  'locality_name':18,
                                  'locality_qualifier':19,
                                  'postcode':20,
                                  'territory':21,
                                  'country':22,
                                  'dob_day':23,
                                  'dob_month':24,
                                  'dob_year':25,
                                  'rec_id':0,
                                  'soc_sec_id':26,
                        missing_values = ['','missing'])

# The following are fields that are passed without standardisation

# The last output field contains the probability of the address HMM

Various types of look-up tables are loaded from their files in the next code block. Not that for example several name tagging look-up table files are loaded into one name tagging look-up table. More details on the file format of these look-up tables and correction lists can be found in Chapter 13.
In this following code block two hidden Markov models (HMMs) are defined (first their states and then the observations, i.e. tags) and loaded from files. These HMMs are used by the name and address component standardisation processes. Further information on HMMs is given in Chapters 7 and 8.
# Define and load hidden Markov models (HMMs)

name_states = ['titl', 'baby', 'knwn', 'andor', 'gname1', 'gname2', 'ghyph',
               'gopbr', 'gclbr', 'agnamel', 'agname2', 'coma', 'sname1',
               'sname2', 'shyph', 'sopbr', 'sclbr', 'asname1', 'asname2',
               'pref1', 'pref2', 'rubb']

name_tags = ['NU', 'AN', 'TI', 'PR', 'GF', 'GM', 'SN', 'ST', 'SP', 'HY', 'CO',
            'NE', 'II', 'BO', 'VB', 'UN', 'RU']

myname_hmm = hmm('Name HMM', name_states, name_tags)
myname_hmm.load_hmm('./hmm/name-absdiscount.hmm')

address_states = ['wfnu', 'wfnal', 'wfna2', 'wfql', 'wfty', 'unnu', 'unty',
                   'prna1', 'prna2', 'inna1', 'inna2', 'inty', 'panu',
                   'pany', 'hyph', 'sla', 'coma', 'opbr', 'clbr', 'loc1',
                   'loc2', 'locq1', 'pc', 'ter1', 'ter2', 'cntr1', 'cntr2',
                   'rubb']

address_tags = ['PC', 'N4', 'NU', 'AN', 'TR', 'CR', 'LN', 'ST', 'IN', 'IT',
                'LQ', 'WT', 'WN', 'UT', 'HY', 'SL', 'CO', 'VB', 'PA', 'UN',
                'RU']

myaddress_hmm = hmm('Address HMM', address_states, address_tags)
myaddress_hmm.load_hmm('./hmm/address-absdiscount.hmm')

Next follows a list of possible date formats needed to parse date strings in the data standardisation process. See Section 6.6 for more details on these format strings.

# Define a list of date parsing format strings

date_parse_formats = ['%d %m %Y', # 24 04 2002 or 24 4 2002
                      '%d %B %Y', # 24 Apr 2002 or 24 April 2002
                      '%m %d %Y', # 04 24 2002 or 4 24 2002
                      '%B %d %Y', # Apr 24 2002 or April 24 2002
                      '%Y %m %d', # 2002 04 24 or 2002 4 24
                      '%Y %B %d', # 2002 Apr 24 or 2002 April 24
                      '%Y%m%d', # 20020424 ** ISO standard **
                      '%d%m%Y', # 24042002
                      '%Y%m%d', # 24042002
                      ']
Now the different standardisation processes can be defined. Each needs input fields and output fields. It is assumed that these input fields are defined in the input data set as defined in the record standardiser further below. Similarly, output fields need to be defined in the output data set of the record standardiser. In this example the output data set of the record standardiser is the temporary data set defined earlier. This temporary data set will later be used as input for the deduplication process.

Each of the component standardiser (for dates, names or addresses) has special arguments which are explained in details in Chapter 6. It is possible to define more than one standardiser, for example if several dates need to be cleaned and standardised, or if more than one name component or several addresses are available in a record.

```python
# Define standardisers for dates
dob_std = DateStandardiser(name = 'DOB-std',
                           description = 'Date of birth standardiser',
                           input_fields = 'date_of_birth',
                           output_fields = ['dob_day', 'dob_month', 'dob_year'],
                           parse_formats = date_parse_formats)
```

Name cleaning and standardisation can be done using a rule-based approach or a hidden Markov model (HMM) approach. The following two code blocks show definitions for both a rule-based and a HMM based name standardisers. For more details on HMM standardisation see Chapter 7, while the name standardisation is explained in Section 6.3 using rules and in Section 6.4 for the HMM based approach.

```python
# Define a standardiser for names based on rules
name_rules_std = NameRulesStandardiser(name = 'Name-Rules',
                                         input_fields = ['given_name',
                                                         'surname'],
                                         output_fields = ['title',
                                                         'gender_guess',
                                                         'given_name',
                                                         'alt_given_name',
                                                         'surname',
                                                         'alt_surname'],
                                         name_corr_list = name_correction_list,
                                         name_tag_table = name_lookup_table,
                                         male_titles = ['mr'],
                                         female_titles = ['ms'],
                                         field_separator = ' ',
                                         check_word_spill = True)
```
# Define a standardiser for name based on HMM

```python
define_hmm_std = NameHMMStandardiser(name = 'Name-HMM',
    input_fields = ['given_name', 'surname'],
    output_fields = ['title', 'gender_guess', 'given_name', 'alt_given_name', 'surname', 'alt.surname'],
    name_corr_list = name_correction_list,
    name_tag_table = name_lookup_table,
    male_titles = ['mr'],
    female_titles = ['ms'],
    name_hmm = myname_hmm,
    field_separator = ' ',
    check_word_spill = True)
```

For addresses, currently only a HMM based standardisation approach is available. See Section 6.5 for more details and a description of all the possible arguments for this standardiser.

```python
# Define a standardiser for address based on HMM

define_address_hmm_std = AddressHMMStandardiser(name = 'Address-HMM',
    input_fields = ['street_num', 'address_part_1', 'address_part_2', 'suburb', 'postcode', 'state'],
    output_fields = ['wayfare_number', 'wayfare_name', 'wayfare_qualifier', 'wayfare_type', 'unit_number', 'unit_type', 'property_name', 'institution_name', 'institution_type', 'postaddress_number', 'postaddress_type', 'locality_name', 'locality_qualifier', 'postcode', 'territory', 'country', 'address_hmm_prob'],
    address_corr_list = addr_correction_list,
    address_tag_table = addr_lookup_table,
    field_separator = ' ',
    address_hmm = myaddress_hmm)
```
In many data sets there are input fields that are already in a cleaned and standardised form, and which therefore can directly be copied into corresponding output fields (so they are available for a linkage or deduplication process later). The simple PassFieldStandardiser as described in Section 6.7 can be used for this. It copies the values from the input fields into the corresponding output fields without any modifications. In the example given below, values from the input field ‘rec_id’ are copied into the output field of the same name, and values from input field ‘soc_sec_id’ are copied into an output field of the same name.

```python
# Define a record standardiser
comp_stand = [dob_std, name_rules_std, address_hmm_std, pass_fields]
```

Now that the component standardisers are defined, they can be passed on to a record standardiser, which defines the input and output data sets for the data cleaning and standardisation process. A check is made to determine if the input and output fields in all the component standardisers defined above are available in the input or output data sets defined in the record standardiser.

As can be seen in the `comp_stand` list, not all defined component standardisers need to be passed to the record standardiser. Only the ones listed here are actually used for the standardisation process. It is very easy to change the standardisation process by simply changing the component standardisers passed to the record standardiser.

```python
example_standardiser = RecordStandardiser(name = 'Example-std',
                                          description = 'Exam. standardiser',
                                          input_dataset = indata,
                                          output_dataset = tmpdata,
                                          comp_std = comp_stand)
```

In the next few code blocks definitions for the record linkage (or deduplication) process are given. First, a blocking index is defined with three different indexes on various field combinations and encoding methods. **Febrl** contains various indexing methods, including the standard blocking, sorted neighbourhood and an experimental bigram index. In the code shown below indexes for all three methods are defined but only one will later be used in the definition of the deduplication process (in this example the blocking index). Note that for a linkage process the index methods used must be the same for both data sets (i.e. it is not possible to use a blocking index for one and data set a sorted neighbourhood index for the other). See Section 9.1 for more details on the various indexing methods and their functionalities.
# Define blocking index(es) (one per temporary data set)

```python
myblock_def = [{('surname','dmetaphone', 4), ('dob_year','direct')},
              [{('given_name','truncate', 3), ('postcode','direct')},
              [{('dob_month','direct'), ('locality_name','nysiis')},
              ]
```

# Define one or more indexes (to be used in the classifier further 
# below)

```python
example_block_index = BlockingIndex(name = 'Index-blocking',
                                     dataset = tmpdata,
                                     index_def = myblock_def)

example_sorting_index = SortingIndex(name = 'Index-sorting',
                                      dataset = tmpdata,
                                      index_def = myblock_def,
                                      window_size = 3)

example_bigram_index = BigramIndex(name = 'Index-bigram',
                                    dataset = tmpdata,
                                    index_def = myblock_def,
                                    threshold = 0.75)
```

Similar to the definition of component standardisers earlier, we now have to define field comparison functions which will be used to compare the selected fields in the cleaned data set in order to calculate the matching weight vector. Section 9.2 describes all the field comparison functions available. The following code block shows different examples of field comparator functions.
# Define comparison functions for linkage

given_name_nysiis = FieldComparatorEncodeString(name = 'Given name NYSIIS',
fields_a = 'given_name',
fields_b = 'given_name',
m_prob = 0.95,
u_prob = 0.001,
missing_weight = 0.0,
encode_method = 'nysiis',
reverse = False)

surname_dmetaphone = FieldComparatorEncodeString(name = 'Surname D-Metaphone',
fields_a = 'surname',
fields_b = 'surname',
m_prob = 0.95,
u_prob = 0.001,
missing_weight = 0.0,
encode_method = 'dmetaphone',
reverse = False)

locality_name_key = FieldComparatorKeyDiff(name = 'Locality name key diff',
fields_a = 'locality_name',
fields_b = 'locality_name',
m_prob = 0.95,
u_prob = 0.001,
missing_weight = 0.0,
max_key_diff = 2)

wayfare_name_winkler = FieldComparatorApproxString(name = 'Wayfare name Winkler',
fields_a = 'wayfare_name',
fields_b = 'wayfare_name',
m_prob = 0.95,
u_prob = 0.001,
missing_weight = 0.0,
compare_method = 'winkler',
min_approx_value = 0.7)

postcode_distance = FieldComparatorDistance(name = 'Postcode distance',
fields_a = 'postcode',
fields_b = 'postcode',
m_prob = 0.95,
u_prob = 0.001,
missing_weight = 0.0,
geocode_table = pc_geocode_table,
max_distance = 50.0)
age = FieldComparatorAge(name = 'Age',
fields_a = ['dob_day', 'dob_month', 'dob_year'],
fields_b = ['dob_day', 'dob_month', 'dob_year'],
m_probability_day = 0.95,
u_probability_day = 0.03333,
m_probability_month = 0.95,
u_probability_month = 0.083,
m_probability_year = 0.95,
u_probability_year = 0.01,
max_perc_diff = 10.0,
fix_date = 'today')

The defined field comparators can now be passed to a record comparator as shown in the following code block. Besides a list of field comparison functions, a record comparator must be given references to the two data sets to be compared. A check will be performed to ensure that the field names given to the field comparison functions (as in the above code blocks) are available in the given data sets.

# ====================================================================
# Define a record comparator using field comparison functions

field_comparisons = [given_name_nysiis, surname_dmetaphone,
locality_name_key, wayfare_name_winkler,
postcode_distance, age]

example_comparator = RecordComparator(tmpdata, tmpdata,
field_comparisons)

The last thing that needs to be defined before a linkage or deduplication can be started is the definition of a classifier that classifies the vectors of weights as calculated by the field comparison functions above into links, non-links or possible links. The available classifiers are described in Section 9.5. The arguments given to the classifier are the two data sets, and in our example – a classical Fellegi and Sunter classifier – the values for the lower and upper thresholds.

# ====================================================================
# Define a classifier for classifing the matching vectors

example_fs_classifier = \
FellegiSunterClassifier(name = 'Fellegi and Sunter',
dataset_a = tmpdata,
dataset_b = tmpdata,
lower_threshold = 0.0,
upper_threshold = 30.0)

Now that all necessary components have been defined and initialised, a deduplication (or similarly a linkage) process can be started easily by invoking the corresponding method of the project object. The various defined components need to be given as arguments, as well as the range of records (first record and number of records) to be deduplicated, and the desired form of outputs. See Section 9.6 for a detailed description of how to start a deduplication or linkage process, and Chapter 10 for descriptions of all possible output forms.
# Start a deduplication task

myproject.deduplicate(input_dataset = indata,
tmp_dataset = tmpdata,
rec_standardiser = example_standardiser,
rec_comparator = example_comparator,
blocking_index = example_block_index,
classifier = example_fs_classifier,
first_record = 0,
number_records = 5000,
output_histogram = 'dedup-example-histogram.res',
output_rec_pair_details = 'dedup-example-details.res',
output_rec_pair_weights = 'dedup-example-weights.res',
output_threshold = 10.0,
output_assignment = 'one2one')

# myfebrl.finalise()

A Febrl project module is properly ended by a call to the finalise() method of your Febrl project object. This will stop the parallel environment (if Febrl has been stated in parallel) and make sure everything is properly shut down.

Note: In the case that an input data set is already available in a cleaned and standardised form, the argument rec_standardiser can be set to None, and no cleaning and standardisation is performed. Instead, the records from the input data set are directly copied into the temporary data set. Therefore, the input data set and the temporary data set must have the same field name definitions.

Once you have modified a project module according to your needs (lets say you edited a file called myproject.py), you can run it by simply typing

    python myproject.py

into your terminal (assuming you are in the correct directory), and Febrl should run according to your settings.
Data Cleaning and Standardisation

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 and standardisation 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.

As an example, the record in Figure 6.1 with four input components is cleaned and split into 14 output fields (the dark gray boxes). Comparing these output fields individually with the corresponding 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 or address from other records.

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. Due to its object-oriented approach, it is fairly easy to integrate custom data transformation procedures written by end users with other aspects of Febrl processing.
The data cleaning and standardisation is implemented in the module `standardisation.py` and uses various routines from the modules `address.py`, `date.py` and `name.py` which contain functionalities to clean and standardise the corresponding components. Different correction lists and tagging look-up tables are used for the cleaning and standardisation tasks.

The following Section 6.1 discusses the underlying techniques used for cleaning and standardisation of names and addresses in more details. A list of the possible output fields is given in Section 6.2, followed by descriptions in Sections 6.3, 6.4, 6.5 and 6.6, of how component standardisers for names, addresses and dates are initialised. Finally, Section 6.8 describes how a record standardiser can be initialised and how records can be cleaned and standardised.

### 6.1 Name and Address Cleaning and Standardisation

**Febrl** data cleaning and standardisation for the name and address components 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 Chapter 7. 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. The process of creating training data is described in Chapter 8. Once HMMs are available for a given data set (or class of data sets), the data cleaning and standardisation process becomes easy and efficient.

The data cleaning and standardisation process for the name and address components in **Febrl** consists of the three steps described in the following sections.

**Febrl** also contains the functionality to check word spilling, i.e. if words in a field are cut off because of limited field length (for example in fixed width input fields) and continue in the next field. Section 6.1.4 explains how word spilling is dealt with.

#### 6.1.1 Step 1: Cleaning

The input to the data cleaning routine is a string that contains an input component, i.e. either a name or an address. First, all letters in such a string are converted into lower case. Then, a correction list of replacement strings is used to replace certain words, abbreviations and characters with others. For example, given the example correction list in Table 6.1, variations of known as, such as ‘a.k.a.’ or ‘aka’ are all replaced with a standard string ‘known as’. A correction list is loaded from a correction list file (see Section 13.1 for the details of the formats of such 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. 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.

Each correction list is sorted and processed by decreasing length of the original string, i.e. long original strings are searched for and replaced first. In the example correction list given below, the entry ‘known as’ 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 thus be converted into ‘ber dette’.

The output of the data cleaning routine is a new string where all occurrences of 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.

#### 6.1.2 Step 2: 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 various look-up tables (and some hard-coded rules), each element of this list is then 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 look-up table and which do not match any of the hard-coded tagging rules are assigned the 'UN' (unknown) tag. 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' (using the example look-up table shown in Table 6.2).

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. As another 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' (assuming this word is not found in any look-up tables for address words).

<table>
<thead>
<tr>
<th>Original</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>'doctor'</td>
<td>'dr'</td>
</tr>
<tr>
<td>'doc'</td>
<td>'dr'</td>
</tr>
<tr>
<td>'md'</td>
<td>'dr'</td>
</tr>
<tr>
<td>'phd'</td>
<td>'dr'</td>
</tr>
<tr>
<td>'miss'</td>
<td>'ms'</td>
</tr>
<tr>
<td>'misses'</td>
<td>'ms'</td>
</tr>
<tr>
<td>'mister'</td>
<td>'mr'</td>
</tr>
</tbody>
</table>

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).

6.1.3 Step 3: Segmentation

Once a word and tag list is available, the tags are used to segment the input word elements into the correct output fields. For names, both a simple rules based and a probabilistic hidden Markov model (HMM) approach are implemented in Febrl (see Appendix C for a description of the rule-based system), but only HMM-based processing is available for addresses (which tend to have more complex and variable layouts and formats than names do). 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 HMM approach is discussed in more details in Chapters 7.

6.1.4 Word Spilling

Word spilling happens when data is entered into input fields with fixed length and continuous typing automatically continues in the next field if a field is full. For example, if a given name field with maximal length of 10 characters is given, and a surname field with 20 characters, the name 'maria louisa miller' would be stored as given name 'maria loui' and surname 'sa miller'. To check for word spilling can be a successful data cleaning step if a data set contains such data.

Word spilling concatenates words at the end and beginning of fields and then checks if such a concatenated word is known, i.e. if it is listed in one of the available look-up tables. If so, the concatenated word is kept, otherwise (i.e. if the word is not known) a whitespace character is inserted between the two original words.

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 Febrl 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 in the data cleaning and standardisation process.

6.2 Output Fields

Currently the Febrl system is capable to clean and standardise three different components. Each of them segments the input (one or more fields from an input data set) into several output fields as shown in Table 6.3. Note that more than one word can be stored into an output field, or it can be empty if the input record did not contain a corresponding input value (e.g. no postcode value).

6.3 Name Cleaning and Standardisation using a Rules Based Approach

Two different methods are currently implemented in the Febrl system to clean and standardise names. A rules based approach is described in this section and a hidden Markov model based approach in the following section. Both are
implemented in the `standardisation.py` and `name.py` modules, and both standardise a name into six name output fields as shown in the first column in Table 6.3.

A rules based name standardiser can be initialised as shown in the code example below. The following arguments need to be given.

- **name**
  A name for the name standardiser. Most suitable is a short string.

- **description**
  A longer description of the name standardiser. Note that this argument is not mandatory, the standardisation process works fine without a description.

- **input_fields**
  A string with a field name or a list of field names from the input data set, that will be standardised. If a list is given the field values will be concatenated (using the `field_separator` as described below) into one string before the parsing and standardisation is done.

- **output_fields**
  A list of six field names as defined in the output data set. The name component standardiser returns standardised names in these six fields, in the sequence as given in Table 6.3. It is possible to set some of these output fields to `None` if no output is to be written (for example if one is not interested in the gender guess values), as long as at least one output field is defined (not set to `None`).

- **name_corr_list**
  A reference to the name correction list to be used. This list must have been initialised and loaded previously. See Chapter 13 and Section 13.1 for more details.

- **name_tag_table**
  A reference to the name tagging table to be used. This table must have been initialised and loaded previously. See Chapter 13 and Section 13.2 for more details.

- **male_titles**
  A list of one or more male title words (like ‘mr’), which will be used to guess the gender.
- **female_titles**
  A list of one or more female title words (like ‘ms’), which will be used to guess the gender.

- **first_name_comp**
  To give the rules based standardisation system a hint, the component names are most likely to start with (either given- or surnames) need to be given using this argument. The value can be either ’gname’ (assuming names start with given names) or ’sname’ (assuming names start with the surname first). The default value (if this argument is not given) is ’gname’.

- **field_separator**
  When more than one input field is given, they are concatenated and a field separator character (which can be an empty string) is inserted between them. The default value for the field separator is a whitespace ’ ’.

- **check_word_spill**
  A flag that can be set to True or False in order to activate or de-activate the word spilling functionality (see Section 6.1.4 for more details). If set to true word spilling between input fields is checked using the name tag look-up table (e.g. ‘peter pa’, ’ul miller’ will be corrected into ’peter paul miller’). Note that word spilling is only useful if the value of the field separator is not an empty string. The default value for word spilling is True.

The following example code shows how a rules based standardiser for names can be initialised. Note that the second output field, i.e. the gender guess, is set to None, which means it will not be stored in the cleaned and standardised data set. Word spilling is activated, and the field separator is set to be a whitespace.

```python
# ====================================================================
name_rules_std = NameRulesStandardiser(name = 'Name-Standard-Rules',
    input_fields = ['gname', 'sname'],
    output_fields = ['title',
                     None,
                     'given_name',
                     'alt_given_name',
                     'surname',
                     'alt.surname'],
    name_corr_list = name_correction_list,
    name_tag_table = name_lookup_table,
    male_titles = ['mr'],
    female_titles = ['ms'],
    field_separator = ' ',
    check_word_spill = True)
```  

### 6.4 Name Cleaning and Standardisation using a Hidden Markov Model Based Approach

The hidden Markov model approach for name cleaning and standardisation is explained in more details in Chapter 7. Similar to the rules based approach, names are standardised into the six name output fields listed in the first column in Table 6.3.

The initialisation of a HMM based name standardiser is very similar to the rules based standardiser. All arguments used for the rules based standardiser can also be used for the HMM based name standardiser. A new additionally argument that needs to be given is
• name_hmm

A reference to a hidden Markov model for names, which must be initialised and loaded. See the example code below on how to initialise and load a HMM.

Assuming the necessary modules have been imported, a hidden Markov model (HMM) can be initialised and loaded as shown in the following code example. A HMM based name standardiser can then be initialised easily.

```python
# ====================================================================
name_states = ['titl','baby','knwn','andor','gname1','gname2','ghyph',
                'gopbr','gclbr','agname1','agname2','coma','snamel',
                'sname2','shyph','sopbr','sclbr','asname1','asname2',
                'pref1','pref2','rubb']

name_tags = ['NU','AN','TI','PR','GF','GM','SN','ST','SP','HY','CO',
             'NE','II','BO','VB','UN','RU']

myname_hmm = hmm('Name HMM', name_states, name_tags)
myname_hmm.load_hmm('./hmm/name-absdiscount.hmm')

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
name_hmm_std = NameHMMStandardiser(name = 'Name-Standard-HMM',
    input_fields = ['gname',
                    'sname'],
    output_fields = ['title',
                     None,
                     'given_name',
                     'alt_given_name',
                     'surname',
                     'alt.surname'],
    name_hmm = myname_hmm,
    name_corr_list = name_correction_list,
    name_tag_table = name_lookup_table,
    male_titles = ['mr'],
    female_titles = ['ms'],
    field_separator = ',
    check_word_spill = True)
```

6.5 Address Cleaning and Standardisation using a Hidden Markov Model Based Approach

Address cleaning and standardisation is currently implemented using a HMM approach only, as described in more details in Chapter 7. Addresses are standardised into the seventeen address output fields as shown in the second column in Table 6.3. The address component standardiser is implemented in the modules standardisation.py and address.py.

A HMM based address standardiser can be initialised as shown in the code example below. The following arguments need to be given.

• name

A name for the address standardiser. Most suitable is a short string.
• **description**
  A longer description of the address standardiser. Note that this argument is not mandatory, the standardisation process works fine without a description.

• **input_fields**
  A string with a field name or a list of field names from the input data set, that will be standardised. If a list is given the field values will be concatenated (using the `field_separator` as described below) into one string before the parsing and standardisation is done.

• **output_fields**
  A list of seventeen field names as defined in the output data set. The address standardiser returns standardised addresses in these seventeen fields, in the sequence as given in the second column in Table 6.3. It is possible to set some of these output fields to `None` if no output is to be written (for example if one is not interested in the country values), as long as at least one output field is defined (not set to `None`). Note the last field which returns the probability returned from the address HMM.

• **address_hmm**
  A reference to a hidden Markov model for addresses, which must be initialised and loaded. See the example code on page 18 on how to initialise and load a HMM.

• **address_corr_list**
  A reference to the address correction list to be used. This list must be initialised and loaded previously. See Chapter 13 and Section 13.1 for more details.

• **address_tag_table**
  A reference to the address tagging table to be used. This table must be initialised and loaded previously. See Chapter 13 and Section 13.2 for more details.

• **field_separator**
  When more than one input field is given, they are concatenated and a field separator character (which can be an empty string) is inserted between them. The default value for the field separator is a whitespace ‘ ’.

• **check_word_spill**
  A flag that can be set to True or False in order to activate or de-activate the word spilling functionality (see Section 6.1.4 for more details). If set to true word spilling between input fields is checked using the address tag look-up table. Note that word spilling is only useful if the value of the `field_separator` is not an empty string. The default value for word spilling is True.

The following example code shows how a HMM based standardiser for addresses can be initialised. Note that the last three output fields for territory, country and the HMM probability are set to `None`, thus no territory and country values will be stored in the output data set, and the HMM probability is not stored either. Word spilling is activated, and the field separator is set to be a whitespace (their default values are taken as both these arguments are not given).
6.6 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. The routines related to date standardisation are implemented in the date.py module.

The aim of date standardisation is to split a given date string into a valid numerical triplet \([\text{day}, \text{month}, \text{year}]\). 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 whitespace.

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 \hspace{1em} For abbreviated month names (Jan, Feb, Mar, etc.)
- \%B \hspace{1em} For full month names (January, February, etc.)
- \%d \hspace{1em} 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 \hspace{1em} For month as a decimal number between 1 and 12.
- \%y \hspace{1em} For year without century as a decimal number, i.e. between 00 and 99 (two digits).
- \%Y \hspace{1em} For year with century as a decimal number (four digits).
- \%U \hspace{1em} For unknown days. If days are unknown a value of ‘unk’ or ‘unknown’ can be parsed.
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 list of date parsing format strings needs to be defined in the project.py module as shown in the example in Chapter 5.

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 is important. The user should order this list 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 an argument to a date standardiser as shown below. The pivot year separates the range of two-digit years 00-99 into two parts, one that is expanded with 19xx (year numbers equal to and after the pivot year), the other with 20xx (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.

When a date standardiser is initialised, the following arguments need to be given.

- name
  A name for the date standardiser. Most suitable is a short string.

- description
  A longer description of the date standardiser. Note that this argument is not mandatory, the standardisation process works fine without a description.

- input_fields
  A string with a field name or a list of field names from the input data set, that will be standardised. If a list is given the field values will be concatenated into one string before the parsing and standardisation is done.

- output_fields
  A list of three field names as defined in the output data set. The date component standardiser returns standardised dates in these three fields, with the day value in the first field, the month value in the second and the year value in the third field. It is possible to set fields to None if no output is to be written to a field (for example if one is not interested in the day values), as long as at least one output field is defined (not None).

- parse_formats
  A string with a date parse format or a list with date parsing format strings as described above.

- pivot_year
  Value of the pivot year (between 00 and 99) that controls expansion of two-digit year values into four-digit year values as described above. A two-digits year value xx smaller than the pivot year will be expanded into 20xx, and year values larger and equal than the pivot year will be expanded into 19xx. The default value is the current year plus one.

The following example code shows how a list of date parsing format strings and two different date standardisers are initialised. They have different input fields and the parsed dates are standardised into different output dates. Note that for the mother standardiser only the year output field is used, both the day and month values are set to None and thus discarded.
date_parse_formats = ['%d %m %Y',  # 24 04 2002 or 24 4 2002
                     '%d %B %Y',  # 24 Apr 2002 or 24 April 2002
                     '%m %d %Y',  # 04 24 2002 or 4 24 2002
                     '%B %d %Y',  # Apr 24 2002 or April 24 2002
                     '%Y %m %d',  # 2002 04 24 or 2002 4 24
                     '%Y %B %d',  # 2002 Apr 24 or 2002 April 24
                     '%Y%m%d',  # 20020424 ** ISO standard **
                     '%d%m%Y',  # 24042002
                    ]

baby_std = DateStandardiser(name = 'Baby-dob-std',
                       description = 'Baby date of birth standardiser',
                       input_fields = 'bdob',
                       output_fields = ['baby_day','baby_month','baby_year'],
                       parse_formats = date_parse_formats)

mother_std = DateStandardiser(name = 'Mother-dob-std',
                       description = 'Mother year of birth stand.',
                       input_fields = 'mdob',
                       output_fields = [None, None, 'mother_year'],
                       parse_formats = date_parse_formats,
                       pivot_year = 04)

6.7 Field Passing

In many data sets there are fields that are already in a cleaned and standardised form, and which therefore can directly be copied into corresponding output fields (so they are available for a linkage or deduplication process later). The simple PassFieldStandardiser can be used for this. It copies the values from the input fields into the corresponding output fields without any modifications. The number of input fields must be the same as the number of output fields. Values in the first input field will be copied into the first output field, values from the second input field into the second output field, and so on. The example code below shows how to apply the field pass standardiser on two fields in a fictitious hospital data set.

The following arguments must be given when a field pass standardiser is initialised.

- **name**
  A name for the field pass standardiser. Most suitable is a short string.

- **description**
  A longer description of the field pass standardiser. Note that this argument is not mandatory.

- **input_fields**
  A string with a field name or a list of field names from the input data set. Values from this input field(s) will be copied into the output field(s). The number of input fields must be the same as the number of output fields.

- **output_fields**
  A string with a field name or a list of field names from the output data set. Values from the input field(s) will be copied into this output field(s). The number of output fields must be the same as the number of input fields.

In the example given in the following code block, values from the input field ‘ohoscode’ are passed (copied) into the output field ‘hosp_code’, and values from input field ‘rseqnum’ are copied into output field ‘seq_num’.
6.8 Record Cleaning and Standardisation

Cleaning and standardising records is done by defining components (like names, addresses and dates) made of one or more fields in the input data set and initialising corresponding component standardisers as explained in the previous sections, resulting in cleaned and standardised output fields. Examples of component and record standardisers were already presented in Chapter 5 and the previous sections. Here, all available arguments for the record standardiser are described and an example is given (see code below). The record standardiser is implemented in the module standardisation.py.

When a record standardiser is initialised, the following arguments must be given.

- **name**
  A name for the record standardiser. Most suitable is a short string.

- **description**
  A longer description of the record standardiser. Note that this argument is not mandatory, the standardisation process works fine without a description.

- **input_dataset**
  A reference to the input data set, from where the fields for the components are taken. Note that all input fields that are defined in component standardisers (as described in the previous sections) must be fields of this input data set.

- **output_dataset**
  A reference to the output data set, which will contain the cleaned and standardised output fields. Note that all output fields that are defined in component standardisers (as described in the previous sections) must be fields of this output data set.

- **component_standardisers**
  This must be a list of one or more component standardisers as presented in the previous sections. As shown in the examples in Chapter 5, these component standardisers must be initialised before they can be passed to a record standardiser using this argument.

Once one or more component standardisers and one record standardiser are initialised, the standardisation process can be started by loading records from the input data set, and passing them on to the record standardiser using the `standardise` (for one record) or `standardise_block` (for a list of records). The following example code shows how to initialise a record standardiser, assuming that the module `standardisation.py` has been imported, and both input and output data sets and several component standardisers have been initialised. Note that the standardisation is normally done within the deduplication and linkage routines as described in Section 9.6. Alternatively, if one is only interested in cleaning and standardising a data set a standardisation process can be started as shown in Section 6.9.
my_comp_standard = [name_rules_stand, address_hmm_stand, baby_std]

my_record_standardiser = RecordStandardiser(name = 'my-rec-stand',
                                           input_dataset = my_in_data,
                                           output_dataset = my_out_data,
                                           component_standardisers = my_comp_standard)

# Load and standardise one record
#
one_record = my_in_data.read_record()

one_clean_record = my_record_standardiser.standardise(one_record)

# Load and standardise a block of records
#
records = my_in_data.read_records(0,10000)  # Load 10,000 records

clean_record_block = my_record_standardiser.standardise_block(records)

6.9 Starting a Standardisation Process

Within a project object one method (routine) is available to define and start a standardisation process. Assuming that a project object has been created (by copying and modifying the template module project-standardise.py) and input and output data sets, as well as component and a record standardisers have been defined, standardisation of a data set can be done by one simple call to the method standardise as shown in the following example.

myproject.standardise(input_dataset = hospital_data,
                      output_dataset = clean_hospital_data,
                      rec_standardiser = hospital_standardiser,
                      first_record = 0,
                      number_records = 100000)

In the given example, 100,000 records in a fictitious hospital data set are standardised and written into an output data set (assuming it has been initialised).

The following arguments need to be defined for the standardisation process.

- **input_dataset**
  A reference to a data set which contains the (raw) input data. This data must be initialised in read access mode.

- **output_dataset**
  A reference to an output data set, which has been initialised in write, readwrite or append access mode. This output data set can be any data set implementation except a memory based data set (as all standardised records would be lost once the program finishes). See Chapter 12 for more information on data set implementations.
• **rec_standardiser**
  A reference to a record standardiser. See Section 6.8 for more details on how to initialise a record standardiser.

• **first_record**
  The record number of the first record in the input data set to be processed. If this argument is not given or set to None (default), the first record (i.e. record with number 0) is taken.

• **number_records**
  The number of records from the input data set that should be processed. If this argument is not given, all records in the input data set are processed.
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-of-heart.

_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\(^1\) [16] 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) [22] 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 [22]. The use of HMMs in data standardisation was the topic of two recent research papers. Borkar et al. [3] 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. [24] 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 7.1: Simple 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.

---

\(^1\) AutoStan and AutoMatch as formerly sold by MatchWare Technologies are now part of the Ascential Integrity (R) product line. See http://www.ascentialsoftware.com
Transition and observation probabilities are stored in two matrices, as shown in the two tables below. The sum of all transition probabilities out of a given state (one row in Table 7.1) has to equal 1.0, as does the sum of all observation probabilities for a particular state.

Figure 7.1 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 7.1: Example name HMM transition probabilities.

<table>
<thead>
<tr>
<th>From state</th>
<th>Start</th>
<th>Title</th>
<th>Givenname</th>
<th>Middlename</th>
<th>Surname</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>–</td>
<td>0.50</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>–</td>
</tr>
<tr>
<td>Title</td>
<td>–</td>
<td>0.0</td>
<td>0.85</td>
<td>0.0</td>
<td>0.10</td>
<td>0.05</td>
</tr>
<tr>
<td>Givenname</td>
<td>–</td>
<td>0.05</td>
<td>0.0</td>
<td>0.25</td>
<td>0.65</td>
<td>0.05</td>
</tr>
<tr>
<td>Middlename</td>
<td>–</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.00</td>
<td>0.0</td>
</tr>
<tr>
<td>Surname</td>
<td>–</td>
<td>0.05</td>
<td>0.20</td>
<td>0.0</td>
<td>0.0</td>
<td>0.75</td>
</tr>
<tr>
<td>End</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 7.2: Example name HMM observation probabilities.

<table>
<thead>
<tr>
<th>Observation</th>
<th>Start</th>
<th>Title</th>
<th>Givenname</th>
<th>Middlename</th>
<th>Surname</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>TI</td>
<td>–</td>
<td>0.96</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>–</td>
</tr>
<tr>
<td>GM</td>
<td>–</td>
<td>0.01</td>
<td>0.35</td>
<td>0.33</td>
<td>0.15</td>
<td>–</td>
</tr>
<tr>
<td>GF</td>
<td>–</td>
<td>0.01</td>
<td>0.35</td>
<td>0.27</td>
<td>0.14</td>
<td>–</td>
</tr>
<tr>
<td>SN</td>
<td>–</td>
<td>0.01</td>
<td>0.09</td>
<td>0.14</td>
<td>0.45</td>
<td>–</td>
</tr>
<tr>
<td>UN</td>
<td>–</td>
<td>0.01</td>
<td>0.20</td>
<td>0.25</td>
<td>0.25</td>
<td>–</td>
</tr>
</tbody>
</table>

In the Febri 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. [3] 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 Febri 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 [22].

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

'\text{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 ‘\text{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

2. ['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 (initial probability), 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

As can be seen, the first tag sequence results in a higher probability, so this one will be selected. 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 selected. 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. Results of experiments both with names and addresses are published in [4, 5].

Thus, with this HMM based standardisation method, 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 8.

7.1 Hidden Markov Model Implementation Module 'simplehmm.py'

The hidden Markov model (HMM) functionalities used in the Febrl system are implemented in the simplehmm.py module. This module provides a class hmm with methods to initialise a HMM, to set its transition and observation
probabilities, to train a HMM, to save it to and load it from a text file, and to apply the Viterbi algorithm to an observation sequence. Additionally, methods to print a HMM (its states, observations symbols all the probabilities) and to check its probabilities (if they sum up correctly in each state) are provided.

For more details we refer the reader to the source code of the simplehmm.py module and its unit testing module simplehmmTest.py. The following example program code (mainly taken from the simplehmmTest.py module) shows how to initialise, train, use, save and load a HMM using the simplehmm.py module. It is assumed that the simplehmm.py module has been imported using the Python command import simplehmm.

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/projects/hmm/)
# Define HMM state list and observation list

test_hmm_states = [‘title’, ‘givenname’, ‘surname’]

# Some example training records (one per line) with state/tag pairs

train_data = [[(‘title’, ‘TI’), (‘givenname’, ‘GF’), (‘surname’, ‘SN’)],
[(‘givenname’, ‘GM’), (‘surname’, ‘UN’)],
[(‘title’, ‘UN’), (‘givenname’, ‘GM’), (‘surname’, ‘UN’)],
[(‘title’, ‘TI’), (‘givenname’, ‘SN’), (‘surname’, ‘SN’)],
[(‘givenname’, ‘GM’), (‘surname’, ‘SN’)],
[(‘title’, ‘TI’), (‘givenname’, ‘GF’), (‘surname’, ‘SN’)],
[(‘surname’, ‘UN’), (‘givenname’, ‘UN’)],
[(‘givenname’, ‘GF’), (‘surname’, ‘GF’), (‘surname’, ‘SN’)]
]

# Some test examples (observation (tag) sequences), one per line

test_data = [[‘TI’, ‘GM’, ‘SN’],
[‘UN’, ‘SN’],
[‘TI’, ‘UN’, ‘UN’],
[‘TI’, ‘GF’, ‘UN’],
[‘UN’, ‘UN’, ‘UN’, ‘UN’],
[‘TI’, ‘GM’, ‘UN’, ‘SN’],
[‘GF’, ‘UN’]]

# Initialise a new HMM and train it

test_hmm = simplehmm.hmm(‘Test HMM’, test_hmm_states, test_hmm_observ)
test_hmm.train(train_data) # Train the HMM

test_hmm.check_prob() # Check its probabilities

test_hmm.print_hmm() # Print it out

# Apply the Viterbi algorithm to each sequence of the test data

for test_rec in test_data:
    [state_sequence, sequence_probability] = test_hmm.viterbi(test_rec)

# Initialise and train a second HMM using the same training data and applying Laplace smoothing

test_hmm2 = simplehmm.hmm(‘Test HMM 2’, test_states, test_observ)
test_hmm2.train(train_data, smoothing=’laplace’)

# Save the second HMM into a text file

test_hmm2.save_hmm(‘testhmm2.hmm’)

# Initialise a third HMM, then load the previously saved HMM into it

test_hmm3 = simplehmm.hmm(‘Test HMM 3’, [‘dummy’], [‘dummy’])
test_hmm3.load_hmm(‘testhmm2.hmm’)
test_hmm3.print_hmm() # Print it out
Hidden Markov Model Training

Note: A clearer, more detailed guide to the hidden Markov model (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 using HMMs as described in Chapter 7 on a new data set, two HMMs – one for names and one for addresses – need to be trained using training records from the same data set which 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 7) 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 tagdata.py module (see Section 8.1 below) provides various options to automatically choose a random sub-set 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 unduly 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 five 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. Each sequence is made of pairs containing an observation symbol (or tag in our case, the uppercase two-letter word), followed by a colon and a HMM state (lower-case entity following the colon). Theses training examples are extracted from the original data set using the tagdata.py program and the HMMs are created using the trainhmm.py (see Section 8.2 below) program. The following is a basic step-by-step guide for hidden Markov model training within Febrl:

1. First, create a file with a small number of training records using the tagdata.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 that are 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 (one line) 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 trainhmm.py using the modified training file. Set the HMM smoothing option hmm_smoothing to either None, laplace or absdiscount. Borkar et al. [3] 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. 1,000 records) again using the tagdata.py program this time set the hmm_file_name to the name of the initial HMM created in the previous step. 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 the 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 option as desired.

7. Create a third training file by reprocessing your second, corrected training file using the tagdata.py program and setting the hmm_file_name to the name of the second HMM file you just created, plus setting the name of the retag_file_name to the name of the 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 option as desired.

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 re-tag 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 your main project module (as derived from an original project.py module), set the name or address HMM file name to the last HMM file you created.

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 setting the freqs_file_name option in tagdata.py 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 tagdata.py when the freqs_file_name option is set – 100,000 would not be unusual. Of course, there is no prospect of being able to inspect all 100,000 records in the resulting training file, but records with unusual patterns amongst those 100,000 records can be found at or near the top of the output file specified with the freqs_file_name option. Corrected versions of the tag:hmm_state pair sequences for those unusual records can then be added to the smaller, 1,000 record training file as discussed above, and the HMM then retrained using the trainhmm.py program.
8.1 Program ‘tagdata.py’

The program tagdata.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 process within the standardisation.py module. 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:

```
python tagdata.py
```

All settings are within the program as shown in the code example at the end of this section. The following list describes the different configuration settings that must be defined.

- After the module header, all necessary modules are imported, and a febrl object is defined. Generally there is no need to modify this.
- A project logger is defined next. The settings like name of the log file, logging and verbose levels, can be modified as described in Chapter 14.
- The input data set (from where the records are selected randomly) needs to be defined as described in Chapter 12. Note that the access mode must be set to 'read'.
- The number of 'num_rec_to_select' records are selected randomly in the range between (and including) records number 'start_rec_number' and 'end_rec_number'. Note that only records with non-empty name or address component are selected.
- The selected and tagged records are written into a file with the name as defined in 'output_file_name'. An example of the format of this output file is given further below.
- The component to be tagged specified with 'tag_component' can be either 'name' or 'address'.
- The names of the fields that constitute the chosen component must be listed in 'tag_component_fields'. Note that these field names must correspond to fields names as defined in the input data set 'fields'.
- If word spilling from a field into another should be checked, then the value of 'check_word_spilling' should be set to True (and the value of 'field_separator' not to an empty string ''), otherwise set this option to False.
- When fields are concatenated to form a component, the 'field_separator' character or string is inserted between them. Note that for word spilling to be activated the values of this field separator must be different from an empty string ''. 
- A hidden Markov model (HMM) file can be defined in the 'hmm_file_name' in which case the training records are tagged as well as standardised 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 or tags for cases that are standardised incorrectly. This mechanism reduces the time needed to create enough records to train a HMM training. If no HMM standardisation is desired, set the value of 'hmm_file_name' to None.
- It is possible to re-process (i.e. re-tag) an existing training file by setting the 'retag_file_name' to the name of an existing training file. This is useful when some adjustments have been made to some tagging look-up tables or correction lists. Note that the re-tagged file will be written to the output file name. Re-tagging is only possible if a HMM file is defined. If no re-tagging should be done, set the value of 'retag_file_name' to None.
- By setting the 'freqs_file_name' option it is possible to compile and write the frequencies of all tag:hmm_state pair sequences in ascending order into the file with the defined 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 can only be used together with a HMM defined.
Finally, tagging look-up tables and correction lists need to be defined. These should be based on the same files as the ones used in a module derived from the `project.py` module, so that cleaning and tagging is done in the same way.

If the option `hmm_file_name` is defined (set to the name of a HMM file), the selected training records are given both tags and (hidden) states, as `tag:hmm_state` pairs, using the HMM. This allows a semi-automatic training process, where the user only has to inspect the output 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 format of the output file is as follows. The selected original input records (name or address 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 a HMM file is defined in option `hmm_file_name` 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
```

The following code example shows the part of the `tagdata.py` program that needs to be modified by the user according to her or his needs.
# Set up Febrl and create a new project (or load a saved project)
tag_febrl = Febrl(description = 'Data tagging Febrl instance',
        febrl_path = '.')
tag_project = tag_febrl.new_project(name = 'Tag-Data',
        description = 'Data tagging module',
        file_name = 'tag.fbr')

# Define a project logger
tag_log = ProjectLog(file_name = 'tag_data.log',
        project = tag_project,
        log_level = 1,
        verbose_level = 2,
        clear_log = True,
        no_warn = False)

# Define settings for data tagging
# Define your original input data set - - - - - - - - - - - - - - - -
input_data= DataSetCSV(name = 'example1in',
        description = 'Example data set 1',
        access_mode = 'read',
        header_lines = 1,
        file_name = './dbgen/dataset1.csv',
        fields = {'rec_id':0,
                'given_name':1,
                'surname':2,
                'street_num':3,
                'address_part_1':4,
                'address_part_2':5,
                'suburb':6,
                'postcode':7,
                'state':8,
                'date_of_birth':9,
                'soc_sec_id':10},
        fields_default = '',
        strip_fields = True,
        missing_values = ['','missing'])

# Define block of records to be used for tagging - - - - - - - - - - - -
start_rec_number= 0
end_rec_number = 1000 # input_data.num_records

# Define number of records to be selected randomly - - - - - - - - - - -
num_rec_to_select= 500

# Define name of output data set - - - - - - - - - - - - - - - - - - - -
output_file_name = 'tagged_data.txt'

8.1. Program 'tagdata.py'
tag_component = 'name'

tag_component_fields = ['given_name', 'surname']

check_word_spilling = True # Set to True or False

field_separator = '

hmm_file_name = './hmm/name.hmm' # Set to name of a HMM file or None

# Retag an existing training file
# - Note that re-tagging is only possible if a HMM file name is given as well
# - If the retag file name is defined, the start and end record numbers as defined above are not used, instead the record numbers in the re tag file are used.
# retag_file_name = None # Set to name of an existing training file or to None

# Write out frequencies into a file
# fregs_file_name = 'fregs.txt' # Set to a file name or None

# Define and load lookup tables
name_lookup_table = TagLookupTable(name = 'Name lookup table',
                                      default = '')
name_lookup_table.load(['.data/givenname_f.tbl',
                        '.data/givenname_m.tbl',
                        '.data/name_prefix.tbl',
                        '.data/name_misc.tbl',
                        '.data/saints.tbl',
                        '.data/surname.tbl',
                        '.data/title.tbl'])

name_correction_list = CorrectionList(name = 'Name correction list')
name_correction_list.load('./data/name_corr.lst')

Chapter 8. Hidden Markov Model Training
address_lookup_table = TagLookupTable(name = 'Address lookup table',
   default = '')

address_lookup_table.load(['.data/country.tbl',
   '.data/address_misc.tbl',
   '.data/address_qual.tbl',
   '.data/institution_type.tbl',
   '.data/post_address.tbl',
   '.data/saints.tbl',
   '.data/territory.tbl',
   '.data/unit_type.tbl',
   '.data/wayfare_type.tbl'])

address_correction_list = CorrectionList(name = 'Address corr. list')

address_correction_list.load('.data/address_corr.lst')


8.2 Program 'trainhmm.py'

Once tagged training data has been created using the program tagdata.py and edited by a user, a hidden Markov model (HMM) can be created using trainhmm.py.

The program is called from the command line with:

   python trainhmm.py

All settings are within the module as shown in the code example at the end of this section. The following list describes the different configuration settings that must be defined.

- After the module header, all necessary modules are imported, and a febrl object is defined. Generally there is no need to modify this.
- A project logger is defined next. The settings like name of the log file, logging and verbose levels, can be modified as described in Chapter 14.
- The name of the input file containing training data must be defined in 'hmm_train_file'. Such a file can be created using the program tagdata.py as described in Section 8.1.
- The trained HMM will be written into a text file with the name as defined in 'hmm_model_file'.
- The HMM can be given a name that describes it using the 'hmm_name' option. Not that this name is not the file name.
- The component 'hmm_component' should be the same as used to tag the training records in 'hmm_train_file'. Possible values are 'name' or 'address'.
- A smoothing method can be chosen using the 'hmm_smoothing' option. Possible values are None for no smoothing, 'laplace' for Laplace smoothing or 'absdiscout' for absolute discount smoothing. Both smoothing methods are described in [3].

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

- Comment lines start with a hash character ('#'). Blank lines are allowed and are 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 (see the example in Section 8.1)

\[ \text{tag:hmm\_state} \]

where the \text{tag} is one of the possible tags as listed in Appendix B, and \text{hmm\_state} is one of the possible states from the state lists in Appendix A (either for the name or the address component). Any unknown tag or state in the training data will result in an error and the program stops.

The following code example shows the part of the \text{trainhmm.py} program that needs to be modified by the user according to her or his needs.

```python
# Set up Febrl and create a new project (or load a saved project)

hmm_febrl = Febrl(description = 'HMM training Febrl instance',
                    febrl_path = '.

hmm_project = hmm_febrl.new_project(name = 'HMM-Train',
                                     description = 'Training module for HMMs',
                                     file_name = 'hmm.fbr')

# Define a project logger

hmm_log = ProjectLog(file_name = 'hmm_train.log',
                      project = hmm_project,
                      log_level = 1,
                      verbose_level = 2,
                      clear_log = True,
                      no_warn = False)

# Define settings for HMM training

# Name of the file containing training records
hmm_train_file = 'tagged_data.txt'

# Name of the HMM file to be written
hmm_model_file = 'test_data.hmm'

# Name of the HMM
hmm_name = 'Test Name HMM'

# Component: Can either be 'name' or 'address'
hmm_component = 'name'

# HMM smoothing method, can be either None, 'laplace' or 'absdiscount'
hmm_smoothing = 'absdiscount'
```

Chapter 8. Hidden Markov Model Training
Record Linkage and Deduplication

Record linkage is the task of comparing records and deciding whether they are a match (i.e. determining if they represent the same entity) or a non-match (i.e. determining if they represent different entities), or, if this decision cannot be made by the record linkage system, using human intervention (clerical review) to decide the matching status of a record pair. Assuming that cleaned and standardised records are available, the process of linking records or the deduplication of a data set consists of several steps.

1. One or more (blocking) indexes need to be built with the aim of grouping together records that potentially match and thus reducing the huge number of possible comparisons. While this grouping should reduce the number of comparisons made as much as possible, it is important that no potential match is overlooked because of the indexing process.

2. After index(es) are built, records within the same index block are compared field by field using field comparison functions, resulting in a weight vector for each record pair compared.

3. These weight vectors are then given to a classifier (like the classical Fellegi and Sunter [8] approach classifier) that decides if a record pair constitutes a match, non-match or a possible match.

The following sections describe in detail how a linkage or deduplication process can be defined using a project object (as shown in the example at the end of Chapter 5), and how its necessary components (such as indexes, field comparison functions and classifiers) can be defined. Indexing is the topic of Section 9.1. All field comparison functions available are described in Section 9.2, and the initialisation of a record comparator is presented in Section 9.3. Section 9.4 includes example code that shows both field comparison functions and record comparator initialisation. The definition of classifiers is then discussed in Section 9.5, and the chapter concludes with Section 9.6 which presents how to define and start a linkage or deduplication process, respectively.

9.1 Indexing

The aim of indexing is to reduce the potentially huge number of comparisons (every record in one data set with all records in another data set) by eliminating comparisons between records that obviously are not matches. In other words, indexing reduces the large search space by forming groups of records that are very likely to be matches. Indexing can also be seen as a clustering method that brings together records that are similar, so only these records need to be compared using the more expensive (i.e. compute intensive) field comparisons functions as explained in Section 9.2 below.

Currently the Febrl system contains several indexing methods, including the traditional blocking method used in many record linkage systems. These indexing methods are implemented in the module indexing.py. Indexes are normally built while a data set is being standardised. After an index is built a compacting has to be done which builds index data structures that can return the blocks more efficiently.

Note: For a linkage process the index methods used must be the same for both data sets (i.e. it is not possible to use a blocking index for one and data set a sorted neighbourhood index for the other).
All indexing methods have the following attributes which can (some need) to be given as arguments when an indexing method is initialised.

- **name**
  A name for an indexing method. This should be a short string.

- **description**
  A longer description for an indexing method. This attribute is not necessary, but can be useful for documenting the indexing method.

- **dataset**
  A reference to the data set the index is built for (i.e. when the index is built it is assumed that records are having the fields as defined in this data set).

- **block_definition**
  The definition of how the index and its blocks should be built. Blocks are defined using a list of lists each containing tuples of the form `(field name, method, parameters)`. The given field names must be available in the defined data set. Methods and parameters are explained in more details in the following subsections.

- **skip_missing**
  A flag, if set to `True` records which have empty indexing values will be skipped over, if set to `False` a block with an empty indexing field value will be included in the index as well. Note that such a block can become quite large if there are many missing values in a data set, extending the run time of a linkage or deduplication process significantly. The default value is `True`.

In the following example, three indexes are defined and a traditional blocking index is initialised. The first index is based on the `Soundex` encoding of the reversed surname field values, with a maximal code length of three, the second index is based on a combination of the first two characters in the given name field (the `truncate` method is used for this) concatenated with the values in the postcode field (the method is `direct` which means the postcode values are not encoded or modified in any way), and the third index is a based on concatenation of the first two digits in the postcode plus the `NYSIIS` encoding of the surname values.

```python
# ........................................................................
hosp_block_def = [['surname','soundex', 3, 'reverse']],
                  [['givenname','truncate',2], ('postcode','direct')],
                  [('postcode','truncate',2), ('surname','nysiis')],
                  
hospital_index = BlockingIndex(name = 'HospIndex',
                               dataset = tmpdata,
                               block_definition = hosp_block_def)
```

As can be seen from the example, an indexing definition is made of a Python list (square brackets), with one or more index definitions inside, each again being a Python list (one per line in the above example). An index definition itself is made of Python tuples of the form `(field_name, method, parameters)`. Field names must be valid field names from the data set the index is applied on (see argument `dataset` in the index definition). The `method` can be set to one of the following methods (with corresponding parameters).

- **direct**
  The field values are directly taken as (part of) the blocking variables. No additional parameter is needed.

- **soundex**
  The `Soundex` phonetic encoding of the values in the given field are taken. A possible first parameter is the
maximal length of the encoding (in characters), and a second parameter can be the word ‘reverse’. When
given, the values in the field are reversed before they are encoded. The default value for the maximal length is
4.

- **mod_soundex**
  A modified version of the Soundex algorithm. The same parameters maximal length and ‘reverse’ as with
  Soundex can be given.

- **nysiis**
  The *NYSIIS* phonetic encoding algorithm. The same parameters maximal length and ‘reverse’ as with
  Soundex can be given.

- **phonex**
  The *Phonex* [14] phonetic encoding algorithm which combines ideas from both Soundex and NYSIIS. The same
  parameters maximal length and ‘reverse’ as with Soundex can be given.

- **dmetaphone**
  The *Double-metaphone* [20] phonetic encoding algorithm. The same parameters maximal length and ‘re-
  verse’ as with Soundex can be given.

- **truncate**
  A string truncation method, where as additional parameter the length must be given (i.e. strings longer than the
  given length are truncated).

Note that all encoding methods are implemented in the module `encode.py`.

In the following sections the available indexing methods are described in more details and their special arguments are
listed.

### 9.1.1 Block Indexing

The traditional blocking index as described in the record linkage literature groups records into blocks if they have the
same values in a blocking variable, like the *Soundex* encoding of the reversed surnames in the example above. See
Section 9.1 above on how to define and initialise a blocking index.

### 9.1.2 Sorting Indexing

The sorting index extends the idea of the classical blocking index in that the values of the blocks (e.g. the *Soundex*
encodings of surnames) are sorted alphabetically and then a sliding window is moved over these sorted blocks. When
a sorting index is initialised, one argument (besides the base class arguments as presented in Section 9.1 above) that
needs to be given is:

- **window_size**
  A positive integer that gives the size of the sliding window.

For example, let’s assume there are six blocks in our index (shown are the blocking variable values and the correspond-
ing record numbers in the blocks), and we have a sliding window size of 3:

```
a123:   [4,12,89,99]
a129:   [6,32,54,84,91]
a245:   [1,39]
a689:   [3,17,21,35,49,76,87,93]
a911:   [2,42,66]
b111:   [8]
```
While with the blocking index only the records within one block are compared with the records in the corresponding block of the second data set, with the sorting index and window size 3 larger blocks are formed by combining three consecutive blocks together:

\[
[a_{123}, a_{129}, a_{245}] : [1, 4, 6, 12, 32, 39, 54, 84, 89, 91, 99] \\
[a_{129}, a_{245}, a_{689}] : [1, 3, 6, 17, 21, 32, 35, 49, 54, 76, 84, 87, 91, 93] \\
[a_{245}, a_{689}, a_{911}] : [1, 2, 3, 17, 21, 35, 39, 42, 49, 66, 76, 87, 93] \\
[a_{689}, a_{911}, b_{111}] : [2, 3, 8, 17, 21, 35, 42, 49, 66, 76, 87, 93]
\]

The idea behind this is, that neighbouring blocks might contain records with similar values in the blocking variables due to errors in the original values. The example below shows how to define and initialise a sorting index.

```python
# ==============================================================
hosp_block_def = [
    [('surname', 'soundex', 3, 'reverse')],
    [('givenname', 'truncate', 2), ('postcode', 'direct')],
    [('postcode', 'truncate', 2), ('surname', 'nysiis')],
]

hospital_index = SortingIndex(name = 'HospIndex',
                              dataset = tmpdata,
                              block_definition = hosp_block_def,
                              window_size = 5)
```

Note that if the window size is set to 1 then the sorting index becomes equivalent to the blocking index.

### 9.1.3 Bigram Indexing

This index implements a data structure based on bigrams and allows for fuzzy blocking. The basic idea is that after an index has been built, the values of the blocking variables will be converted into a list of bigrams, and permutations of sub-lists will be built using a given threshold (a number between 0.0 and 1.0) of all possible permutations. The resulting bigram lists will be inserted into an inverted index, i.e. record numbers in the blocks will be inserted into Python dictionaries for each bigram. Such an inverted index will then be used to retrieve the blocks.

When a bigram index is initialised, one argument (besides the base class arguments as presented in Section 9.1 above) that needs to be given is:

- threshold
  A number between 0.0 (not including) and 1.0 (including).

For example, assume a block definition contains the tuple

```python
block_definition = [\[('givname','direct')\], ...]
```

and the bigram threshold is set to 0.8. If a value 'peter' is given in the 'givname' (given name) field, the corresponding bigram list will be ['pe','et','te','er'] with four elements, so using the 0.8 threshold results in \(4 \times 0.8 = 3.2\) rounded to 3, which means all permutations of length 3 are calculated. For the given example they are

```python
[‘pe’,’et’,’te’]
[‘pe’,’et’,’er’]
[‘pe’,’te’,’er’]
[‘et’,’te’,’er’]
```

So, the corresponding record number will be inserted into the inverted index blocks with keys ‘peette’, ‘peeter’, ‘peteer’, and ‘etteer’.
The lower the threshold, the shorter the sub-lists, but the more sub-lists there will be per field value, resulting in more (smaller blocks) in the inverted index.

The following example shows how to define and initialise a bigram index.

```python
# ====================================================================

hosp_block_def = [(['surname','soundex', 3, 'reverse']),
                 [('givenname','truncate',2), ('postcode','direct')],
                 [('postcode','truncate',2), ('surname','nysiis')]

hospital_index = BigramIndex(name = 'HospIndex',
                             dataset = tmpdata,
                             block_definition = hosp_block_def,
                             threshold = 0.75)
```

### 9.2 Field Comparison Functions

The heart of the record linkage process consists of the comparison of fields from individual records. These field comparison functions return the basic matching weights that are stored in a weight vector for each record pair that is compared. The weight vectors are then given to a classifier – like the classical Fellegi and Sunter [8] approach – to calculate a matching decision (match, non-match or possible match).

The Febrl system contains a number of different FieldComparator functions, implemented in the module comparison.py. The field comparators allow various comparisons of strings, numbers, dates, ages and times. The following arguments need to be given to all field comparison functions when they are initialised:

- **dataset_a**
  A reference to a data set. Records taken from this data set will be compared to records from **dataset_b**.

- **dataset_b**
  A reference to a data set. This can be the same data set as **dataset_a**, for example if a deduplication is performed. Records taken from this data set will be compared to records from **dataset_a**.

- **fields_a**
  A single field name (as string) or a list of field names from **dataset_a**. If a list of field names is given, the comparison function will concatenate them into one single string (without whitespaces between the fields), and this string will then be compared with a string formed similarly using the fields from **fields_b**.

- **fields_b**
  A single field name (a string) or a list of field names from **dataset_b**. If a list of field names is given, the comparison function will concatenate them into one single string (without whitespaces between the fields), and this string will then be compared with a string formed similarly using the fields from **fields_a**.

- **missing_weight**
  A numerical value (floating-point number) that will be returned if one or more of the fields in **fields_a** or **fields_b** correspond to a missing value (as defined in a data set, see Chapter 12). The default value (i.e. if no argument **missing_weight** is given) for the missing value weight is zero.

- **M-probability**
  The probability that the two fields (or field lists) **fields_a** and **fields_b** are the same in matched record pairs. For most field comparison functions this is a numerical value given as argument **m_probability**. For the date and age comparators separate probabilities need to be given for day, month and year in the three arguments **m_probability_day**, **m_probability_month** and **m_probability_year**.
• U-probability
  The probability that two fields (or field lists) \( \text{fields}_a \) and \( \text{fields}_b \) are the same in un-matched record pairs. The argument names are similar to the ones for the M-probability.

The values for all M- and U-probabilities must be between 0.0 and 1.0. It is possible to re-set the probabilities for a field comparator at any time using the method \texttt{set\_probabilities()}.

The \textit{agreement} and \textit{disagreement} weights are computed using the M- and U-probabilities, as described in the record linkage literature (see for example [8, 10, 16, 27]).

\[
\begin{align*}
\text{agree\_weight} &= \log_2 \left( \frac{m\_probability}{u\_probability} \right) \\
\text{disagree\_weight} &= \log_2 \left( \frac{1.0 - m\_probability}{1.0 - u\_probability} \right)
\end{align*}
\]

Frequency look-up tables can be used with several of the field comparison functions, using the following optional arguments. The calculation of frequency dependent weights is described in detail in Section 9.2.1 below.

- \texttt{frequency\_table}
  A reference to a frequency table (as defined and loaded using methods from the \texttt{lookup.py} module).

- \texttt{freq\_table\_max\_weight}
  A numerical maximal weight value, which can be used to restrict weights computed using frequency tables to a certain limit (for rare entries in large tables weights can become very large and thus dominate other weights totally, which should be prevented).

- \texttt{freq\_table\_min\_weight}
  A numerical minimal weight value, which can be used to restrict weights computed using frequency tables to a certain limit.

Two additional arguments to all field comparison functions are \texttt{name} and \texttt{description} which can be used to document the functionality of a field comparator.

### 9.2.1 Frequency Dependent Weight Calculation

If a frequency table is given for a certain field comparator that supports frequency dependent weight calculation, both agreement and disagreement weight will be calculated using the frequencies of the values of the input fields that are compared, if they are found in the frequency table.

Given a field value is listed in the given frequency table, its count and the sum of all entries in the frequency table are used to compute the frequency probability of this entry.

\[
\text{value\_freq\_prob} = \frac{\text{value\_count}}{\sum \text{value\_count}}
\]

The agreement and disagreement weight are then calculated using the following two formulas.

\[
\begin{align*}
\text{agree\_weight} &= \log_2 \left( \frac{1.0}{\text{value\_freq\_prob}} \right) \\
\text{disagree\_weight} &= \log_2 \left( \frac{1.0 - \text{value\_freq\_prob}}{1.0 - \text{value\_freq\_prob}^2} \right)
\end{align*}
\]
If the attribute \texttt{freq\_table\_max\_weight} (see Section 9.2 above) is set and the calculated agreement weight is larger than this value, it is limited to the value of \texttt{freq\_table\_max\_weight}. Similarly, if the \texttt{freq\_table\_min\_weight} attribute is given and the calculated disagreement weight is smaller than this value, the disagreement weight will be limited to the value of \texttt{freq\_table\_min\_weight}.

If a value is not found in a frequency table, the M- and U-probabilities are used to compute generic agreement and disagreement weights as described in Section 9.2.

For each field value (one from each record) we now have an agreement and a disagreement weight, and the minimum of the two agreement weights will be selected if the values are the same, and the maximum of the disagreement weight if the two values differ. Partial agreement weights are then calculated as described in the sections below.

\textbf{Note:} The choice of using the minimum for agreement weight and maximum for disagreement weight is not based on a sound theory, rather the authors didn’t find enough evidence in the record linkage literature of how to calculate the weights if the values partially agree, and if they have different agreement and disagreement weights.

The following sections contain descriptions of the field comparison functions currently provided by \texttt{Febrl}. Improved and additional functions will be added in later versions of this software.

\section*{9.2.2 Exact String Comparison \texttt{FieldComparatorExactString}}

This field comparator function compares the two fields (or field lists) given to it as strings and returns the agreement weight if they are the same and the disagreement weight if they differ.

If a frequency table is given for this field comparator, both agreement and disagreement weights will be calculated using the frequencies of the values in the input fields that are compared, as described in Section 9.2.1.

\section*{9.2.3 Truncated String Comparison \texttt{FieldComparatorTruncateString}}

This field comparison function allows the comparison of strings that can be truncated at a certain position using the argument \texttt{max\_string\_length}. Only the first \texttt{max\_string\_length} characters in both strings are compared. Similar to the exact string comparison function, this field comparator compares the two fields (or field lists) given to it as strings and returns the agreement weight if the truncated strings are the same and the disagreement weight if they differ.

If a frequency table is given for this field comparator, both agreement and disagreement weights will be calculated using the frequencies of the values in the input fields that are compared, as described in Section 9.2.1.

\section*{9.2.4 Approximate String Comparison \texttt{FieldComparatorApproxString}}

Approximate string comparison is an important feature for successful weight calculation when comparing strings from names and addresses. Instead of simply having an agreement or disagreement weight returned, approximate string comparators allow for partial agreement if strings are not exactly but almost the same, which can be due to typographical and other errors.

Various algorithms for approximate string comparisons have been developed, in both the statistical record linkage \cite{21} and in the computer science and natural language processing communities. In the \texttt{Febrl} system, several approximate string comparison algorithms are implemented in the module \texttt{stringcmp.py}. All string comparison functions implemented in this module return a value between 0.0 (two strings are completely different) and 1.0 (two strings are the same).

The approximate string comparison method has to be selected with the \texttt{compare\_method} argument. The following methods are currently implemented:

\begin{itemize}
  \item \texttt{jaro}
    \begin{itemize}
      \item The \texttt{Jaro} \cite{21, 27} string comparator is commonly used in record linkage software. It computes 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.

- **winkler**
  The Winkler comparator is based on the Jaro comparator but 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. The partial agreement weight is therefore increased if the beginning of two strings is the same.

- **bigram**
  Bigrams are the two-character substrings in a string, for example ‘peter’ contains the bigrams ‘pe’, ‘et’, ‘te’ and ‘er’. In the Bigram string comparator, the number of common bigrams in the two strings is counted and divided by the average number of bigrams in the two strings, to calculate a similarity measure between 0.0 and 1.0.

- **editdist**
  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. This number is then divided by the length of the longer string to get a similarity measure between 0.0 and 1.0.

- **seqmatch**
  This approximate string comparator is implemented in the Python standard library `difflib`. It is based on an algorithm developed by Ratcliff and Obershelp in the 1980s, and uses pattern matching to compute a similarity measure between 0.0 and 1.0.

A second argument that needs to be given to the approximate string comparator is `min_approx_value` (a number between 0.0 and 1.0), which is the minimal approximate string similarity measure tolerated.

If the two strings are the same (i.e. if the similarity measure returned by the approximate string comparator is 1.0), the agreement weight is returned. If the value is less than 1.0 but larger or equal to `min_approx_value`, then a partial agreement weight is calculated using the following formula.

\[
\text{partial\_agreement} = \text{agree\_weight} - \left( \frac{1.0 - \text{similar\_measure}}{1.0 - \text{min\_approx\_value}} \right) \cdot \\
(\text{agree\_weight} + \text{abs}(\text{disagree\_weight}))
\]

If the returned value is smaller than `min_approx_value` the disagreement weight will be returned.

If a frequency table is given for this field comparator, both agreement and disagreement weights will be calculated using the frequencies of the values in the input fields that are compared, as described in Section 9.2.1.

### 9.2.5 Encoded String Comparison `FieldComparatorEncodeString`

Phonetic name encoding is traditionally used to create blocking variables in the record linkage process, but it can also be used to compare strings. Several algorithms for phonetic encoding are implemented in the `encode.py` module.

The encoded string comparison function compares the two fields (or field lists) given to it as encoded strings, and returns the agreement weight if both strings are encoded the same way, otherwise the disagreement weight is returned.

The encoding method has to be selected with the `encode\_method` argument. The following methods are currently implemented in *Febrl*:

- **soundex**
  The Soundex encoding keeps the original first character of a string and calculates a 3-digit code of the remainder of the string.
• **mod_sounds**: Based on the original Soundex encoding this modified encoding algorithm uses a different mapping of letters into numbers.

• **nysiis**: The NYSIIS (New York State Identification Intelligence System) encoding returns a code that contains only characters.

• **phonex**: Phonex [14] is a variation of Soundex and NYSIIS which tries to improve the encoding quality by preprocessing names before they are encoded. Similar to Soundex the code consists of a leading letter followed by numbers.

• **dmetaphone**: The more recently developed algorithm Double-Metaphone [20] attempts to better account for non-English words, like European and Asian names. Similar to NYSIIS, Double-Metaphone returns a code only consisting of letters. In general, Double-Metaphone seems to be closer to the correct pronunciation of names than NYSIIS.

All of these encodings are particularly sensitive to errors in the first letter of a string. Therefore, an additional argument to the encoded string comparator is `reverse` which can be either set to `False` or `True`. In the latter case, the strings are reversed first before they are encoded. The default value for `reverse` is `False`.

The maximum length of the codes calculated can be set with the argument `max_code_length`, which has a default value of 4.

If a frequency table is given for this field comparator, both agreement and disagreement weights will be calculated using the frequencies of the values in the input fields that are compared, as described in Section 9.2.1.

### 9.2.6 Keying Difference Comparison 'FieldComparatorKeyDiff'

This field comparator compares the two fields (or field lists) given to it as strings character-wise, with a maximum number of different characters that can be tolerated. This number has to be set with the argument `max_key_diff`. If the number of different characters is larger than zero but equal to or smaller than `max_key_diff` the partial agreement weight is calculated using the following formula.

\[
\text{partial agreement} = \text{agree weight} - \left( \frac{\text{num key diff}}{\text{max key diff} + 1} \right) \times \left( \text{agree weight} + \text{abs(disagree weight)} \right)
\]

If the number of key differences is larger than `max_key_diff` then the disagreement weight is returned.

This field comparator can also be used to compare numerical fields, such as date of birth, telephone numbers, etc.

If a frequency table is given for this field comparator, both agreement and disagreement weights will be calculated using the frequencies of the value in the input fields that are compared, as described in Section 9.2.1.

### 9.2.7 Numeric Comparison with Percentage Tolerance 'FieldComparatorNumericPerc'

This field comparator is for numeric fields, where a given maximal percentage difference can be tolerated. This percentage has to be set by the argument `max_perc_diff` as a value between 0.0 and 100.0. The default value is 0.0, i.e. no difference is tolerated.

The agreement weight is returned if the numbers are the same, and the disagreement weight if the percentage difference is larger than the `max_perc_diff` value. If the percentage difference between the two values is larger than 0.0 but equal to or smaller than `max_perc_diff` the partial agreement weight is calculated according to the following
where the percentage difference between the two values is calculated as

\[
perc_{\text{diff}} = \left( \frac{\text{abs}(\text{value}_A - \text{value}_B)}{\min(\text{value}_A, \text{value}_B)} \right) \times 100
\]

In its current version no frequency dependent weight calculation is possible, thus the arguments \text{frequency_table}, \text{freq_table_max_weight} and \text{freq_table_min_weight} can not be used with this field comparator function.

### 9.2.8 Numeric Comparison with Absolute Tolerance \('FieldComparatorNumericAbs'\)

This field comparator is for numeric fields, where a given maximal absolute difference can be tolerated. This difference has to be set by the argument \text{max_abs_diff} as a positive value. The default value is 0.0, i.e. no difference is tolerated.

The agreement weight is returned if the numbers are the same, and the disagreement weight if the absolute difference is larger than the \text{max_abs_diff} value. If the absolute difference between the two values is larger than 0.0 but equal to or smaller than \text{max_abs_diff} the partial agreement weight is calculated according to the following formula.

\[
partial_{\text{agreement}} = \text{agree_weight} - \left( \frac{\text{abs_diff}}{\text{max_abs_diff} + 1.0} \right) \times (\text{agree_weight} + \text{abs}(\text{disagree_weight}))
\]

where the absolute difference between the two values is calculated as

\[
\text{abs_diff} = \text{abs}(\text{value}_A - \text{value}_B).
\]

In its current version no frequency dependent weight calculation is possible, thus the arguments \text{frequency_table}, \text{freq_table_max_weight} and \text{freq_table_min_weight} can not be used with this field comparator function.

### 9.2.9 Date Comparison with Day Tolerance \('FieldComparatorDate'\)

This field comparator function compares two dates which must be given in three separate fields \text{day}, \text{month} and \text{year}. If the value in the fields are not valid dates, an error is triggered.

Two additional arguments can be given to this field comparator, namely \text{max_day_a_before_b} which is the maximal tolerated number of days that the date from data set A can be before the date from data set B, and \text{max_day_b_before_a} which is the maximal tolerated number of days that the date from data set B can be before the date from data set A. The default value for both arguments is 0 days, which means only exact agreement is tolerated and all comparisons of different dates result in the disagreement weight being returned.

If Day A is equal to or less than \text{max_day_a_before_b} days before day B, then a partial agreement weight is calculated using the following formula.

\[
partial_{\text{agreement}} = \text{agree_weight} - \left( \frac{\text{date_diff}}{\text{max_day_a_before_b} + 1} \right) \times (\text{agree_weight} + \text{abs}(\text{disagree_weight}))
\]
The case where day A is after day B is calculated similarly using the value of argument \( \text{max\_day\_b\_before\_a} \).

If the day difference between the days is larger than the maximum of \( \text{max\_day\_a\_before\_b} \) and \( \text{max\_day\_b\_before\_a} \) then the disagreement weight will be returned.

**Note:** Separate M- and U-probabilities have to be given for day, month and year. The general \( m\_\text{probability} \) and \( u\_\text{probability} \) arguments can not be used, instead use

\[
\begin{align*}
\text{m\_probability\_day} \\
\text{u\_probability\_day} \\
\text{m\_probability\_month} \\
\text{u\_probability\_month} \\
\text{m\_probability\_year} \\
\text{u\_probability\_year}
\end{align*}
\]

In its current version no frequency dependent weight calculation is possible, thus the arguments \( \text{frequency\_table}, \text{freq\_table\_max\_weight} \) and \( \text{freq\_table\_min\_weight} \) can not be used with this field comparator function.

### 9.2.10 Age Comparison with Percentage Tolerance 'FieldComparatorAge'

This field comparator compares ages which can be given either as a number (assumed to be an age in years) or as dates in three separate fields day, month and year. If these fields are not a valid date, an error is triggered.

Dates are converted into ages relative to a fix date that can be set with the argument \( \text{fix\_date} \). If no fix date is set, ages are calculated relative to the current system date (today). The \( \text{fix\_date} \) argument can either be given as a date string, as a date tuple (in the form \([\text{day},\text{month},\text{year}]\)) or as the string 'today'.

The comparator allows for a certain percentage difference between two ages. The argument \( \text{max\_perc\_diff} \) can be used which sets the maximum tolerated age difference in percentage. The value must be a number between 0.0 and 100.0. The default is 0.0, which means no age difference is tolerated.

If the percentage difference between two ages is smaller than or equal to the \( \text{max\_perc\_diff} \), a partial agreement weight is calculated using the following formula.

\[
\text{partial\_agreement} = \text{agree\_weight} - \left( \frac{\text{perc\_diff}}{\text{max\_perc\_diff} + 1} \right) \times \left( \text{agree\_weight} + \text{abs}\left(\text{disagree\_weight}\right)\right)
\]

where the percentage difference between the two ages is calculated as

\[
\text{perc\_diff} = \left( \frac{\text{abs}(\text{age\_A} - \text{age\_B})}{\text{min}(\text{age\_A}, \text{age\_B})} \right) \times 100.0
\]

If the percentage difference between the two ages is larger than the \( \text{max\_perc\_diff} \) the disagreement weight is returned.

**Note:** Separate M- and U-probabilities have to be given for day, month and year. The general \( m\_\text{probability} \) and \( u\_\text{probability} \) arguments can not be used, instead use

\[
\begin{align*}
\text{m\_probability\_day} \\
\text{u\_probability\_day} \\
\text{m\_probability\_month} \\
\text{u\_probability\_month} \\
\text{m\_probability\_year} \\
\text{u\_probability\_year}
\end{align*}
\]

In its current version no frequency dependent weight calculation is possible, thus the arguments \( \text{frequency\_table}, \text{freq\_table\_max\_weight} \) and \( \text{freq\_table\_min\_weight} \) can not be used with this field comparator function.
9.2.11 Time Comparison with Minute Tolerance 'FieldComparatorTime'

The time field comparator assumes times are in 24-hours format given either as \texttt{HHMM} or \texttt{HH:MM}, with midnight being \texttt{00:00} and noon being \texttt{12:00}. If no valid time is given an error is triggered.

Similar to the day comparator, it is possible to set two tolerance values for time A being before time B and vice-versa with the two arguments \texttt{max\_time\_a\_before\_b} and \texttt{max\_time\_b\_before\_a}, respectively. The value of these arguments must be in minutes between 0 and 1440 (24 hours). Default value for both arguments is 0, which results in exact time comparison only (i.e. if the times are not the same the disagreement value will be returned).

Additionally, it is possible to set the \texttt{day\_start} of a 24-hours period to a time different from midnight using the \texttt{day\_start} (which must be of the form \texttt{HHMM} or \texttt{HH:MM}). It is assumed that both times are within the same 24-hours period. The default value for \texttt{day\_start} is midnight (00:00).

If time A is equal to or less than \texttt{max\_time\_a\_before\_b} before time B the partial agreement will be calculated as follows:

\[
\text{partial\_agreement} = \text{agree\_weight} - \left( \frac{\text{time\_diff}}{\text{max\_time\_a\_before\_b} + 1} \right) \times \left( \frac{\text{agree\_weight} + \text{abs}(\text{disagree\_weight})}{\text{max\_time\_a\_before\_b} + 1} \right)
\]

The case where time A is after time B is calculated similarly using the value of argument \texttt{max\_time\_b\_before\_a}.

If the time difference between the two times is larger than the maximum of \texttt{max\_time\_a\_before\_b} and \texttt{max\_time\_b\_before\_a} then the disagreement weight will be returned.

In its current version no frequency dependent weight calculation is possible, thus the arguments \texttt{frequency\_table}, \texttt{freq\_table\_max\_weight} and \texttt{freq\_table\_min\_weight} can not be used with this field comparator function.

9.2.12 Distance Comparison with Kilometer Tolerance 'FieldComparatorDistance'

Using locality information from external sources, distance comparison can be calculated when postcodes or similar fields are available in data sets. This field comparator uses a geocode look-up table (where for each entry longitude and latitude are stored) to calculate the distance in kilometers between two values. The argument \texttt{geocode\_table} must be set to a reference to a geocode look-up table as defined and loaded using methods from the \texttt{lookup.py} module (see Chapter 13 for more details).

An additional argument is \texttt{max\_distance} (a positive number in kilometers) which gives the maximum distance that is tolerated. If the calculated distance between two fields is equal to or less than the value in \texttt{max\_distance}, a partial agreement weight is calculated using the following formula.

\[
\text{partial\_agreement} = \text{agree\_weight} - \left( \frac{\text{distance}}{\text{max\_distance} + 1.0} \right) \times \left( \frac{\text{agree\_weight} + \text{abs}(\text{disagree\_weight})}{\text{max\_distance} + 1.0} \right)
\]

If the distance is larger than \texttt{max\_distance}, the disagreement weight is returned.

In its current version no frequency dependent weight calculation is possible, thus the arguments \texttt{frequency\_table}, \texttt{freq\_table\_max\_weight} and \texttt{freq\_table\_min\_weight} can not be used with this field comparator function.

9.3 Record Comparator

A \texttt{RecordComparator} is constructed using a list of field comparators. Once field comparison functions have been initialised, they can be inserted into a \texttt{field\_comparisons} list which is then given to a record comparator.
comparing records in a linkage or deduplication process, each field comparator in a field comparisons list will calculate a weight, and all weights for a record pair will be stored in a weight vector. Additional information stored with a weight vector are the unique record identifiers for the two records in the pair compared. The weight vectors will then be used to compute the matching status of the record pair in the classifier as described in Section 9.5.

The `RecordComparator` is initialised using references to two data sets to be linked (which can be the same in case of a deduplication task), and a list of field comparison functions. An example initialisation is given in the following section.

### 9.4 Example Field and Record Comparator Initialisation

Field comparator functions need to be initialised (or constructed) in order to be able to use them. In the following examples, we assume that a data set `mydata_1` has the fields `givenname`, `surname`, `age` and `postcode`, and a second data set `mydata_2` has the fields `gname`, `sname`, `dob`, `mob`, `yob` (day, month and year of birth) and `pcode`. Additionally, a frequency table `surname_freq` is available and has been loaded. The following examples illustrate how to set up different field comparators that can be used to compute weight vectors using a record comparator (which is initialised at the end of the example code).

```python
# ====================================================================

surname_exact = FieldComparatorExactString(fields_a='surname',
                                          fields_b='sname',
                                          m_prob=0.95, u_prob=0.001,
                                          missing_weight=0.0,
                                          frequency_table=surname_freq,
                                          freq_table_max_weight=20.0,
                                          freq_table_min_weight=-20.0)

surname_jaro = FieldComparatorApproxString(fields_a='surname',
                                          fields_b='sname',
                                          m_prob=0.95, u_prob=0.001,
                                          missing_weight=0.0,
                                          frequency_table=surname_freq,
                                          freq_table_max_weight=20.0,
                                          freq_table_min_weight=-20.0,
                                          compare_method='jaro')

givenname_trunc = FieldComparatorTruncateString(fields_a='givenname',
                                          fields_b='gname',
                                          m_prob=0.90, u_prob=0.02,
                                          missing_weight=0.0,
                                          max_string_len=4)

postcode_keydiff = FieldComparatorKeyDiff(fields_a='postcode',
                                          fields_b='pcode',
                                          m_prob=0.98, u_prob=0.001,
                                          missing_weight=0.0,
                                          max_key_diff=1)
```

---

**9.4. Example Field and Record Comparator Initialisation**
9.5 Classification

The last step in a record linkage process – after records have been compared and weight vectors have been calculated – is the classification of record pairs into links, non-links, or if this decision should be done by a human review, possible links.

The current version of Febrl contains the classical Fellegi and Sunter classifier [8], as well as a flexible classifier that allows various ways of calculation of a matching weight. They are discussed in the following two sections.

**Note:** Future version of Febrl will include improved classification methods, based on sophisticated machine learning techniques. In fact, one of the main aims of our future research will be the development of better classification methods with the ultimate aim to reduce – or even eliminate – the class of possible links (and therefore reducing the time consuming clerical review process). For more information please visit the Febrl web site or contact the authors.

9.5.1 Fellegi and Sunter Classifier

The classical Fellegi and Sunter classifier [8] simply sums all the weights in a weight vector (as calculated by a RecordComparator as discussed in Section 9.3), and then uses two thresholds to classify a record pair into one of the three classes links, non-links or possible links. The results of a classification are stored in a data structure, which can then be used to produce various output forms as presented in Chapter 10.

When a Fellegi and Sunter classifier is initialised, the following arguments need to be given.

- name
  A name for a the classifier. This should be a short string.
- **dataset_a**
  A reference to a data set. This must be the same data set as the first data set defined within a record comparator.

- **dataset_b**
  A reference to a data set, which must be the same as the second data set defined within a record comparator. This data set will be the same as `dataset_a` in a deduplication process, but most likely a different data set in a linkage process (until different parts of the same data set are to be linked).

- **lower_threshold**
  A number, which is the lower threshold for the classifier.

- **upper_threshold**
  A number, which is the upper threshold for the classifier. It must be equal to or larger than the lower threshold.

The following example shows how a Fellegi and Sunter classifier can be initialised.

```python
# =============
f_s_classifier = FellegiSunterClassifier(
    name = 'My F & S classifier',
    dataset_a = mydata_1,
    dataset_b = mydata_2,
    lower_threshold = 10.0,
    upper_threshold = 50.0)
```

### 9.5.2 Flexible Classifier

This flexible classifier allows different methods to be used to calculate the final matching weight for a weight vector (as calculated by a `Record Comparator` as discussed in Section 9.3). Similar to the Fellegi and Sunter classifier, two thresholds are used to classify a record pair into one of the three classes links, non-links or possible links. The results of a classification are stored in a data structure, which can then be used to produce various output forms as presented in Chapter 10.

Instead of simply summing all weights in a weight vector, this flexible classifier allows a flexible definition of the final weight calculation by defining tuples containing a function and elements of the weight vector upon which the function is applied. The final weight is then calculated using another function that needs to be defined by the user.

The following functions can currently be used within the flexible classifier:

- **min**
  Take the minimum value in the selected weight vector elements.

- **max**
  Take the maximum value in the selected weight vector elements.

- **add**
  Add the values in the selected weight vector elements.

- **mult**
  Multiply the values in the selected weight vector elements.

- **avrg**
  Calculate the average of the values in the selected weight vector elements.
Weight vector elements are selected by giving the desired indexes (starting from 0) in a Python list, e.g. \([0, 1, 4]\) selects the first two and the fifth field comparison weights. When initialising a flexible classifier, the argument `calculate` needs to be set to a list made of tuples with functions and weight vector elements as shown in the example below.

The final weight can then be calculated by again using one of the functions 'min', 'max', 'add', 'mult', and 'avrg' given above. The argument `final_funct` has to be used for this when a flexible classifier is initialised.

Let's make an example. Assume we have weight vectors that contain weights calculated by eight different field comparison functions (as explained in Section 9.2). We would like to calculate the final weight as being the average of 1) the sum of the first four weights, 2) the maximal value of weights five and six, and 3) the minimum of weights seven and eight. The corresponding flexible classifier can then be initialised as shown in the following example code.

```python
flex_classifier = FlexibleClassifier(name = 'My flexible classifier',
    dataset_a = mydata_1,
    dataset_b = mydata_2,
    lower_threshold = 10.0,
    upper_threshold = 50.0,
    calculate = [('add', [0,1,2,3]),
                 ('max', [4,5]),
                 ('min', [6,7])],
    final_funct = 'avrg')
```

Note that it is possible to use a weight in more than just one of the calculated intermediate weights. Alternatively it is also possible not to use a weight. It is important though that weight vectors must have as much elements as are used in the `calculate` definitions (i.e. one should not use definitions with indexes larger than the lengths of the weight vectors).

When a flexible classifier is initialised, the following arguments need to be given.

- **name**
  - A name for the classifier. This should be a short string.

- **dataset_a**
  - A reference to a data set. This must be the same data set as the first data set defined within a record comparator.

- **dataset_b**
  - A reference to a data set, which must be the same as the second data set defined within a record comparator. This data set will be the same as `dataset_a` in a deduplication process, but most likely a different data set in a linkage process (until different parts of the same data set are to be linked).

- **lower_threshold**
  - A number, which is the lower threshold for the classifier.

- **upper_threshold**
  - A number, which is the upper threshold for the classifier. It must be larger than the lower threshold.

- **calculate**
  - The definitions for the calculation of intermediate results using selected elements of the weight vector. This must be a list containing tuples, with each tuple being made of a function (one of 'min', 'max', 'add', 'mult', or 'avrg') and a list of the weight vector elements to be used (index numbers starting with 0).

- **final_weight**
  - The function to be used to calculate the final weight. Must be one of 'min', 'max', 'add', 'mult', or 'avrg'.
9.6 Starting a Linkage or Deduplication Process

Two methods (routines) to define and start a linkage or a deduplication process are available within a project object, respectively.

Assuming that a project object has been created (by copying either the template module project-linkage.py or project-deduplicate.py provided – see Chapter 5 for more details) and component and record standardisers, field and record comparators, indexes and a classifier have all been defined, deduplication of a data set or linkage of two data sets can be done by one simple call to the corresponding method as shown in the following examples.

```python
# ====================================================================
myproject.deduplicate(input_dataset = hospital_data,
                      tmp_dataset = tmpdata,
                      rec_standardiser = hospital_standardiser,
                      rec_comparator = hospital_comparator,
                      blocking_index = hospital_index,
                      classifier = hospital_fell_sunter_classifier,
                      first_record = 0,
                      number_records = 100000,
                      output_histogram = True,
                      output_rec_pair_details = 'hosp-dedupl-details.txt',
                      output_rec_pair_weights = 'hosp-dedupl-weights.txt',
                      output_threshold = 30.0,
                      output_assignment = 'one2one')
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
myproject.link(input_dataset_a = hospital_data,
               tmp_dataset_a = tmpdata1,
               input_dataset_b = accident_data,
               tmp_dataset_b = tmpdata2,
               rec_standardiser_a = hospital_standardiser,
               rec_standardiser_b = accident_standardiser,
               rec_comparator = hosp_acc_comparator,
               blocking_index_a = hospital_index,
               blocking_index_b = accident_index,
               classifier = hosp_acc_fell_sunter_classifier,
               first_record_a = 0,
               number_records_a = 10000,
               first_record_b = 0,
               number_records_b = 20000,
               output_histogram = True,
               output_rec_pair_details = True,
               output_rec_pair_weights = 'hosp-linkage-weights.txt',
               output_threshold = 40.0,
               output_assignment = 'one2one')
```

In the first example, the records with number 0 to 100,000 in a fictitious hospital data set are deduplicated, and in the second example the records with number 0 to 10,000 in a hospital data set are linked with records number 0 to 20,000 in an accident data set.

For the deduplication method, the following arguments need to be defined.

- **input_dataset**
  A reference to a data set which contains the (raw) input data. This data must be initialised in read access mode.
**tmp_dataset**  
A reference to a direct random access data set (initialised in access mode `readwrite`) that will hold the cleaned and standardised records before they are deduplicated.

**rec_standardiser**  
If the input data set contains records that need to be cleaned and standardised, then this argument should be a reference to a record standardiser that performs this task. If this argument is set to `None` (i.e. `rec_standardiser = None`) then the records from the input data set will be used directly in the deduplication process (they will be directly copied into the temporary data set. In such a case, the temporary data set must have the same field name definitions as the input data set). See Chapter 6 and Section 6.8 for more details on how to initialise a record standardiser.

**rec_comparator**  
A reference to a record comparator that contains field comparison functions that compare record pairs from the temporary data set field by field and that produces a weight vector for each record pair, which is given to the classifier. See Section 9.3 on how to define a record comparator.

**blocking_index**  
A reference to an indexing object defined on the temporary data set as described in Section 9.1.

**classifier**  
A reference to a classifier that classifies weight vectors. See Section 9.5 for more details on classification of weight vectors.

**first_record**  
The record number of the first record in the input data set to be processed. If this argument is not given (or set to `None`), then it will be set to the first record number (i.e. record with number 0).

**number_records**  
The number of records from the input data set that should be processed. If this argument is not given (or set to `None`), it will be set to the total number of records in the data set.

**output_histogram**  
**output_rec_pair_details**  
**output_rec_pair_weights**  
**output_threshold**  
**output_assignment**  
These arguments are needed to define the desired output forms, and are discussed in detail in Chapter 10.

For a record linkage process, similar arguments are needed. The main differences are basically that references to two input and two temporary data sets must be given, then references to two record standardisers and two indexing definitions (one per data set), plus first record and number of records for two data sets.
Output

Several output forms are possible with the current Febrl version, with more to be included in the future. Currently, the three possible output forms are

- printing (displaying) or saving into a text file a histogram of the record pair comparisons weights,
- printing (displaying) or saving into a text file in detail all record pairs subject to a minimal weight threshold, and
- printing (displaying) or saving into a text file all record identification numbers of all compared record pairs and their corresponding weights.

It is possible to select any combination of these three output forms. Additionally, it is possible to activate a one-to-one assignment restriction procedure before the output is generated (but after record pairs were classified). This will restrict each record in a data set to be linked to maximal one record in the other data set (or for a deduplication, a record can be a duplicate of only one other record). See Section 10.1 below for more details on this topic.

Setting the desired output forms as well as activating an assignment restriction procedure needs to be done with the appropriate arguments when the linkage or deduplication process is defined (see Section 9.6). The following output related arguments need to be defined within a deduplication or link method.

- output_histogram
  This argument can be set to True (in which case a histogram is printed – or displayed – in the terminal window), to a file name (a string) (in which case the histogram is written into this file), or to False (in which case no histogram is printed or saved). Such a histogram is made of simple characters, with the weights (starting with low values) being on the vertical axis. For each weight, a bar is displayed horizontally indicating the number of record pairs with this weight, as shown in the following example histogram.

```
Weight histogram:
---------------
0 ***** 47
1 ****************** 199
2 ********** 122
3 **** 34
4 ***** 48
5 ********** 81
6 ****************** 178
7 ******************* 256
8 ****************** 165
9 **** 41
10 ** 38
```
• output_rec_pair_details
This argument can be set to True, to a file name or to False. If set to True all record pairs with a weight larger than output_threshold (if defined, see below) are printed (displayed) in the terminal window. If set to a file name (a string) then the record pairs will be written into this text file. Each record pair is printed or saved in a three column format, with the field names in the first column, then the values of the first record in the medium column, and the values of the second record in the third column. The first line for a record pair is the total weight, and the second line are the two record identifiers (made of the data set names and the record numbers), followed by a number of lines containing the record fields. If a record pair was assigned in the assignment procedure, it will be displayed/saved accordingly (with the string ' [assigned]'), as shown in the example output below (note that names and addresses are selected randomly and are not part of a real world data set).

Resulting record pairs:
-----------------------------------
Output threshold:  10.000000
Data set A: example4a-tmp
Data set B: example4b-tmp

-----------------------------------------------
Weight: 64.378553 [assigned]
Fields | [RecID A: 4973/example4a-tmp] | [RecID B: 330/example4b-tmp]
address_hmm_ | 1.71249174813e-09 | 1.61564691479e-11
dob_day | 28 | 28
dob_month | 12 | 12
dob_year | 1939 | 1939
gender_guess | female | female
given_name | bridget | bridget
locality_nam | parkinson | parkinson
postcode | 2705 | 2705
rec_id | rec-3023-org | rec-3023-dup-0
soc_sec_id | 3815665 | 3815665
surname | hand | hand
territory | new_south_wales | new_south_wales
wayfare_name | bunton | bunton
wayfare_numb | 210 | 210
wayfare_type | place | place

-----------------------------------------------
Weight: 40.274501
Fields | [RecID A: 3228/example4a-tmp] | [RecID B: 763/example4b-tmp]
address_hmm_ | 6.42884838319e-07 | 6.42884838319e-07
dob_day | 11 | 11
dob_month | 11 | 11
dob_year | 1952 | 1952
gender_guess | male |
given_name | edward | edwafd
locality_nam | lykabetos cabramatta | bobblegigbie cabramatta
postcode | 3201 | 3201
rec_id | rec-3179-org | rec-3179-dup-0
soc_sec_id | 5673646 | 5673646
surname | marshman | marshman
territory | western_australia | western_australia
wayfare_name | cambridge | cambridge
wayfare_numb | 4 | 852
wayfare_type | street | street
- **output_rec_pair_weights**
  
  Again, this argument can be set to True, to a file name (a string) or to False. If set to True or to a file name, all record pairs with a weight larger than **output_threshold** (if defined) are displayed (or written into the file). Only the record numbers and the corresponding total weight will be displayed or saved. Additionally, if a record pair has been assigned in the assignment procedure, the string ‘[assigned]’ will be displayed/saved as well. As can be seen in the example below, the first column are the record identification numbers from the first data set and the second column contains the record identification numbers of the matching records in the second data set.

  Resulting record pairs:
  -----------------------
  Output threshold: 10.000000
  Data set A: example4a-tmp
  Data set B: example4b-tmp
  
  0
  1449: 19.9035825896 [assigned]

  1
  2750: 40.2745006027 [assigned]

  2
  4656: 45.0119233965 [assigned]
  777: 11.5367909784

  3
  4119: 59.6411298934 [assigned]
  3306: 49.7493461902

  5
  2305: 24.3405844039
  3944: 23.4012517298 [assigned]

  6
  2801: 54.486768984 [assigned]
  3289: 11.2032464488
  2743: 10.8222053532

- **output_threshold**
  
  If this argument is set to None all record pairs stored in the results data structure (as calculated by the classifier) will be printed or saved. Note that only the record pairs with matching weights larger than the lower threshold (i.e. matches and possible matches) are stored in this results data structure. However, this can still be a huge number of record pairs, many having a low weight. By setting this output threshold to a positive number (e.g. equal to the upper threshold as defined in a Fellegi and Sunter classifier) one can restrict the volume of the output. With the exception of the histogram, all other output forms (as well as assignment procedures) will be restricted to record pairs with weights that are equal to or larger than the output threshold.

- **output_assignment**
  
  This argument can be used if one wants to apply special assignment restrictions, for example if a record can be linked to only one other record. Currently, the only implemented procedure is a one-to-one assignment, which can be selected by setting this argument to ‘one2one’. See Section 10.1 for more information on assignment restrictions.

An example of how to use these arguments when defining a linkage or deduplication process can be seen in Chapter 5 and Section 9.6.
In many linkage projects one is often only interested in the best record pair matches, and one-to-one assignments need to be forced. The simplest way to do this would be to use a greedy algorithm, but this would result in some assignments being not optimal due to the transitive closure problem.

For example, assume a record $A$ is linked to a record $B$ with a match weight of 42.21, and record $A$ is also linked to a record $C$ with a weight of 39.01, and record $B$ is linked to record $C$ with a weight of 44.98. Assuming that the record pair list is sorted, a greedy assignment procedure will assign record $A$ to record $B$ (because the weight for pair $(A, B)$ is larger than for $(A, C)$), but then it can not assign record $C$ to record $A$ because record $A$ has already been assigned (with a lower weight than the $A$ to $C$ weight). Even if the record pair list is sorted according to the match weights the assignment will most likely not be optimal.

This linear sum assignment problem can be solved with special algorithms that find the optimal solution over all possible assignments. In Febrl we have implemented the Auction algorithm (in the lap.py module) as developed by Bertsekas [2] and others.

Our linear assignment procedure implementation takes as input the results calculated from a classifier (i.e. record pairs and their total weights), and optionally a threshold value (in which case all record pairs that have a weight lower than this threshold are not considered in the assignment procedure). A third argument is the process type which can be either deduplication or linkage. The reason for this is that in a deduplication process a record number can only be part of one assignment (i.e. record $X$ in two record pairs $(X, Y)$ and $(Z, X)$ is the same), while for a linkage process a record number $X$ in the first data set does not identify the same record as a record number $X$ in the second data set (i.e. record pair $(X, Y)$ and $(Z, X)$ can both be assigned).

The first step in the assignment procedure (after filtering out all record pairs with weights less than the given threshold) is to find record numbers that are independent, i.e. that do only appear in exactly one record pair, as they are not part of a multi-record assignment problem. Secondly, we extract all sub-sets of connected record pairs that build one assignment problem. Each of these sub-sets can be solved independently, which is then done in the third step using the Auction algorithm [2].

The output of the one-to-one assignment routine is a dictionary that contains the optimal record pair assignments and their corresponding matching weights.
11.1 Program ‘randomselect.py’

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 assumes that each record corresponds to one line.

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

\[
\text{python randomselect.py in\_file out\_file -perc percentage\_value}
\]

or

\[
\text{python randomselect.py in\_file out\_file -num num\_records}
\]

The needed arguments are

- \textit{in\_file}
  
  Name of the input file with the original data records.

- \textit{out\_file}
  
  Name of the output file where the randomly selected records are written into.

- \textit{-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.

- \textit{-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.

11.2 Database generator Program ‘generate.py’

As record linkage is dealing with data sets that contain partially identified data, like names and addresses, it can be very difficult to test and evaluate newly developed software for record linkage. For the user it can be difficult to learn how to apply, evaluate and customise record linkage software effectively without example data sets where the linkage (or deduplication) status is known.

To overcome this we have developed a database generator based on ideas by Hernandez [12]. This generator can create data sets that contain names (based on frequency look-up tables for surnames and given names), addresses (based on frequency tables for suburbs, postcodes, street number and addresses, and state or territory names), dates (like date of birth), and identifier numbers (like a social security number).
This database generator and all its associated files (frequency files, a README.txt documentation, and several example data sets) are part of the Febrl distribution and are stored in the directory dbgen/ within the main Febrl directory. Please read the README.txt documentation in dbgen/ for more information on the content of this directory.

In a first step a number of original records are generated, and in a second step duplicates of these original records are created using randomly introduced modifications. Each of these records is given a unique identifier, which allows the evaluation of accuracy and error rates (false linked record pairs and un-linked true matches) of deduplication and linkage procedures.

Duplicates are created by randomly introducing modifications (with user definable probabilities) of the following forms:

- Insert a new character at a random position into a field (attribute).
- Delete a character at a random position from a field.
- Substitute a character in a field with another character (with the possibility to set probabilities for randomly choosing a neighbouring key in the same keyboard row or column).
- Transpose two characters at a random position in a field.
- Swap (replace) the value in a field with another value (randomly selected from a list of possible values).
- Insert a space into a field and splitting a word.
- Delete a space (if available) in a field and merging two words.
- Set a field value to missing (with a user definable missing value).
- Swap the values of two fields (e.g. the surname with the given name values) in one record.

The generate.py program can be started from the command line with the following argument list

```
python generate.py output_file num_records num_duplicates max_duplicate_per_record distribution
```

The needed arguments are

- **output_file**
  The name of the output file. Currently this is a comma separated values (CSV) text file.

- **num_records**
  The number of original records to be generated. These records are given a unique record identifier of the form ‘rec=X-org’ with X being a record number starting from 0.

- **num_duplicates**
  The number of duplicate records to be generated. These records are given a unique record identifier on the form ‘rec=X-dup-Y’ with X being the record number of the original record upon which a duplicate is based on, and Y being a duplicate sequence number (starting from 0), i.e. if more than one duplicate record is created based on one original record, these duplicates are numbered from 0 onwards.

- **max_duplicate_per_record**
  The maximal number of duplicate records that will be created based on one original record.

- **distribution**
  The probability distribution used to create duplicate records (i.e. the number of duplicates for one original record). Possible distributions are: uniform, poisson, or zipf.
Many more parameters (like all the different probabilities of how modifications are introduced, and the names of the frequency look-up table files to be used for names and addresses) have to be edited within the `generate.py` module itself as shown in the following code section given below (which is directly taken from `generate.py`).

```python
# Set this flag to True for verbose output, otherwise to False
VERBOS_OUTPUT = True

# For each field (attribute), a dictionary has to be defined with the following keys (probabilities can have values between 0.0 and 1.0):
# - name The field name to be used when a header is written into the output file.
# - type The type of the field. Possible are:
#    'freq' (for fields that use a frequency table with field values).
#    'date' (for date fields in a certain range).
#    'iden' (for numerical identifier fields in a certain range).
# - char_range The range of random characters that can be introduced. Can be one of 'alpha', 'digit', or 'alphanum'.
# - freq_file The name of a frequency file (for fields of type 'freq' only).
# - start_date A start date (for fields of type 'date' only), must be a tuple (day,month,year).
# - end_date A end date (for fields of type 'date' only), must be a tuple (day,month,year).
# - start_id A start identification number (for fields of type 'iden' only).
# - end_id A end identification number (for fields of type 'iden' only).
# - ins_prob Probability to insert a character into a field.
# - del_prob Probability to delete a character from a field.
# - sub_prob Probability to substitute a character in a field with another character.
# - trans_prob Probability to transpose two characters in a field.
# - val_swap_prob Probability to swap the value in a field with another (randomly selected) value for this field (taken from this field’s look-up table).
# - spc_ins_prob Probability to insert a space into a field (thus splitting a word).
# - spc_del_prob Probability to delete a space (if available) in a field (and thus merging two words).
# - miss_prob Probability to set a field value to missing (empty).
```
givenname_dict = {'name': 'given_name',
                 'type': 'freq',
                 'char_range': 'alpha',
                 'freq_file': 'data/givenname.csv',
                 'ins_prob': 0.03,
                 'del_prob': 0.04,
                 'sub_prob': 0.05,
                 'trans_prob': 0.03,
                 'val_swap_prob': 0.08,
                 'spc_ins_prob': 0.01,
                 'spc_del_prob': 0.00,
                 'miss_prob': 0.02}

surname_dict = {'name': 'surname',
                 'type': 'freq',
                 'char_range': 'alpha',
                 'freq_file': 'data/surname.csv',
                 'ins_prob': 0.05,
                 'del_prob': 0.04,
                 'sub_prob': 0.06,
                 'trans_prob': 0.05,
                 'val_swap_prob': 0.05,
                 'spc_ins_prob': 0.01,
                 'spc_del_prob': 0.01,
                 'miss_prob': 0.01}

streetnumber_dict = {'name': 'street_num',
                      'type': 'freq',
                      'char_range': 'digit',
                      'freq_file': 'data/streetnumber.csv',
                      'ins_prob': 0.0,
                      'del_prob': 0.01,
                      'sub_prob': 0.0,
                      'trans_prob': 0.02,
                      'val_swap_prob': 0.10,
                      'spc_ins_prob': 0.0,
                      'spc_del_prob': 0.0,
                      'miss_prob': 0.03}

address1_dict = {'name': 'address_1',
                  'type': 'freq',
                  'char_range': 'alpha',
                  'freq_file': 'data/address1.csv',
                  'ins_prob': 0.05,
                  'del_prob': 0.05,
                  'sub_prob': 0.07,
                  'trans_prob': 0.05,
                  'val_swap_prob': 0.02,
                  'spc_ins_prob': 0.05,
                  'spc_del_prob': 0.05,
                  'miss_prob': 0.02}
address2_dict = {'name': 'address_2',
                 'type': 'freq',
                 'char_range': 'alpha',
                 'freq_file': 'data/address2.csv',
                 'ins_prob': 0.04,
                 'del_prob': 0.04,
                 'sub_prob': 0.08,
                 'trans_prob': 0.10,
                 'val_swap_prob': 0.01,
                 'spc_ins_prob': 0.10,
                 'spc_del_prob': 0.05,
                 'miss_prob': 0.09}

suburb_dict = {'name': 'suburb',
                'type': 'freq',
                'char_range': 'alpha',
                'freq_file': 'data/suburb.csv',
                'ins_prob': 0.02,
                'del_prob': 0.03,
                'sub_prob': 0.07,
                'trans_prob': 0.05,
                'val_swap_prob': 0.05,
                'spc_ins_prob': 0.02,
                'spc_del_prob': 0.01,
                'miss_prob': 0.01}

postcode_dict = {'name': 'postcode',
                 'type': 'freq',
                 'char_range': 'digit',
                 'freq_file': 'data/postcode.csv',
                 'ins_prob': 0.00,
                 'del_prob': 0.00,
                 'sub_prob': 0.05,
                 'trans_prob': 0.10,
                 'val_swap_prob': 0.01,
                 'spc_ins_prob': 0.0,
                 'spc_del_prob': 0.0,
                 'miss_prob': 0.0}

state_dict = {'name': 'state',
              'type': 'freq',
              'char_range': 'alpha',
              'freq_file': 'data/state.csv',
              'ins_prob': 0.0,
              'del_prob': 0.0,
              'sub_prob': 0.01,
              'trans_prob': 0.01,
              'val_swap_prob': 0.02,
              'spc_ins_prob': 0.0,
              'spc_del_prob': 0.0,
              'miss_prob': 0.01}
dob_dict = {'name': 'date_of_birth',
            'type': 'date',
            'char_range': 'digit',
            'start_date': (01, 01, 1900),
            'end_date': (31, 12, 1999),
            'ins_prob': 0.0,
            'del_prob': 0.0,
            'sub_prob': 0.01,
            'trans_prob': 0.01,
            'val_swap_prob': 0.04,
            'spc_ins_prob': 0.0,
            'spc_del_prob': 0.0,
            'miss_prob': 0.02}

ssid_dict = {'name': 'soc_sec_id',
             'type': 'iden',
             'char_range': 'digit',
             'start_id': 10000000,
             'end_id': 99999999,
             'ins_prob': 0.0,
             'del_prob': 0.0,
             'sub_prob': 0.02,
             'trans_prob': 0.03,
             'val_swap_prob': 0.04,
             'spc_ins_prob': 0.0,
             'spc_del_prob': 0.0,
             'miss_prob': 0.00}

# --------------------------------------------------------------------
# Now add all field dictionaries into a list according to how they
# should be saved in the output file
field_list = [givenname_dict, surname_dict, streetnumber_dict,
              address1_dict, address2_dict, suburb_dict,
              postcode_dict, state_dict, dob_dict, ssid_dict]
# --------------------------------------------------------------------
# Flag for writing a header line (keys 'name' of field dictionaries)
save_header = True  # Set to 'False' if no header should be written
# --------------------------------------------------------------------
# Probabilities (between 0.0 and 1.0) for swapping values between two
# fields.
# Use field names as defined in the field directories (keys 'name').
field_swap_prob = {('address_1', 'address_2'): 0.05,
                   ('given_name', 'surname'): 0.07}
# --------------------------------------------------------------------
# Probabilities (between 0.0 and 1.0) for creating a typographical
# error (a new character) in the same row or the same column.
# This is used in the random selection of a new character in the
# 'sub_prob' (substitution of a character in a field)
single_typo_prob = {'same_row': 0.4,
                    'same_col': 0.3}
Lets make an example and create a very small data set with 10 original records and 10 duplicate records (with a maximal of 4 duplicate records being created using one original record). A poisson distribution should be used for the creation of duplicates, and the resulting 20 records should written into the text file mydata.txt. The following call of generate.py will create such a data set.

```
python generate.py mydata.txt 10 10 4 poisson
```

The output file mydata.txt will contain a header line with the field names (as defined with the key ‘name’ in the field dictionaries as given above), followed by the 20 records in an unsorted (random) sequence.

The content of mydata.txt is given below. For easier viewing the records have been sorted (using the Unix command sort) so that original and duplicate records are listed together.

```plaintext
id, given_name, surname, street_num, address_1, address_2, suburb, postcode, state, date_of_birth, soc_sec_id
rec-0-dup-0, sybella, white, 11, wyllyplace, inverpine ret vill, tar ee, 4860, sa, 19690705, 8112687
rec-0-dup-1, sybella, carmody, 11, inverpine ret vill, wylly place, taree, 4860, sa, 19690705, 8112678
rec-0-dup-2, sybella, carmody, 3, wylly place, inverpine ret vill, tared, 4860, sa, 19690705, 8112678
rec-0-dup-3, carmody, sybella, 11, wylly place, inverpine ret vill, taree, 4860, sa, 19690705, 8112678
rec-0-org, sybella, carmody, 11, wylly place, inverpine ret vill, taree, 4860, sa, 19690705, 8112678
rec-1-org, lewis, salehi, 15, stuart street, hartford, mentone, 3135, nsw, 19390215, 4314779
rec-2-dup-0, joel, prak, 20, griffith sstreet, myross, st kilda east, 2756, vic, 19490316, 1872693
rec-2-dup-1, joel, beganovic, 20, griffith street, myross, st kilda east, 2756, vic, 19490316, 1872693
rec-2-org, joel, beganovic, 20, griffiths street, myross, st kilda east, 2756, vic, 19490316, 1872693
rec-3-org, erin, denholm, 5, ellenborough street, elura homestead, bundaberg, 4350, nsw, 19320706, 5471990
rec-4-dup-0, white, shakirah, 6, cherry hvaen, dacombocourt, newcaslte, 2089, vic, 19080704, 6529770
rec-4-org, shakirah, white, 6, dacomb court, cherry haven, newcastle, 2089, vic, 19080704, 6529770
rec-5-dup-0, michael, campnaa, , eldridge street, elm brook, eight mile plains, 2040, qld, 19310122, 9989150
rec-5-dup-1, michael, campana, 21, eldridgestreet, , eight mil e plains, 2040, qld, 19310122, 9989150
rec-5-dup-2, michael, campana, 12, eldridge street, elm brook, wollongong north, 2040, qld, 19310122, 9989250
rec-5-org, michael, campana, 21, eldridge street, elm brook, eight mile plains, 2040, qld, 19310122, 9989150
rec-6-org, luke, white, 15, witt place, fir 1 the professional centre, new farm, 2304, qld, 19600207, 4416617
rec-7-org, caitlin, goonan, 35, burkitt street, , parramatta, 2024, vic, 19031018, 4987722
rec-8-org, thomas, green, 20, tinderry circuit, electorate office, emerald, 3040, nsw, 19880220, 4303596
rec-9-org, mason, matthews, 19, tadgell place, , parkville, 6010, nsw, , 2073052
```
CHAPTER

TWELVE

Data Set Access

Data sets can be available in many different formats, often stored either as portable text files or in databases, but sometimes also in efficient non-portable binary files, or in application specific file formats. Within Febrl, the module dataset.py provides the means to access various data set formats. The module has been written to make it easy to add additional methods to read new data formats in the future.

Generally, one can distinguish between data sets that only allow sequential record access, and data sets that allow direct random access to any record stored (using a unique record identifier). While sequential data sets are mainly used for persistent storage of data sets (like in text files or databases), direct random access data sets are very useful for temporary storage – for example after records have been cleaned and standardised and before they are used in a deduplication or linkage process.

The Febrl system currently supports the following five data set implementations.

- COL
  Text files with fields with fixed column width (sequential access).
- CSV
  Comma separated values text files (sequential access).
- SQL
  SQL database access using the open source relational database system MySQL\(^1\) (sequential access).
- Shelve
  Efficient file-based hash-tables (direct random access) provided by Python.
- Memory
  Efficient in-memory hash-tables (direct random access) provided by Python.

Records within a data set in Febrl are internally identified by unique integer numbers (starting with zero). These numbers are not stored in data sets with sequential access, but are used in data sets with direct random access to identify records.

Data sets in Febrl need to be initialised. This merely means that the Febrl program needs to do whatever is necessary to prepare to read or write records to the data sets. A data set can be initialised in one of several access modes, and all further access is restricted to this access mode (for example, a data set initialised for read access can not be written to). This allows one to have input data sets with read access only, temporary data sets with read and write access, and output data data sets with write or append access restrictions.

The following access modes are supported and can be set using the argument access_mode when a data set is initialised.

- read
  For reading records from a data set. If an empty data set is initialised in read mode an error is triggered.

\(^1\) see http://www.mysql.com
- **write**
  For writing to a data set (no reading is possible). If a data set already exists, its content will be erased first. It is not possible to read records from a data set initialised in write access mode.

- **appending**
  For appending records to a data set (i.e. if it already contains records they will not be erased). It is not possible to read records from a data set initialised in append access mode. Note that append mode can only be used with sequential access data sets.

- **readwrite**
  For both reading from and writing to a data set. This access mode is only possible with direct random access data sets.

Each record in a data set comprises one or more **fields**. A Python dictionary with the field names as keys needs to be given when a data set is initialised, as can be seen in the examples in the following subsections. The values in this dictionary depend on the data set type (e.g. for a CSV data set the column numbers need to be given, while for an SQL data set the database column names are needed). For both the shelve and memory data sets only the field names are used, so the values in the fields dictionary can be anything.

It is not necessary to define all fields in a data set. Only the ones that are used in a data cleaning and standardisation or in a record linkage process need to be listed. For example, a data set can contain names and addresses of patients, plus many medical attributes which are not used in a record linkage project (see the various examples throughout this chapter), in which case only the used name and address fields need to be defined in the fields dictionary.

When reading from a data set, records are returned as Python dictionaries, i.e. key:value pairs, where the keys are the field names and the values the corresponding values read from the data set. Note that only non-empty fields are returned, thus such a record dictionary can be quite small even if a data set has many fields.

Similarly, when writing records into a data set, they also have to be in a dictionary with the field names in the data set as keys. For fields that are not stored in a dictionary, a fields_default value will be written into the data set (for sequential access data sets only).

When reading from a data set, for each record two hidden fields are returned, namely _rec_num_ (the record identifier number) and _dataset_name_ (the name of the data set the record is read from). Using these hidden fields a record can be uniquely identified. Note that for direct access data sets the hidden field _rec_num_ is used as a record identifier when records are written into such a data set.

All data set implementations have the following attributes, which can be set with the corresponding key word (argument) when a data set is initialised.

- **name**
  The name of the data set. This should be a short string only.

- **description**
  A longer description of the data set. Note that this argument is not mandatory.

- **access_mode**
  A data set can be initialised in one of the four access modes 'read', 'write', 'append' (sequential access data sets only) or 'readwrite' (direct random access data sets only) as explained above.

- **fields**
  A dictionary with the fields (columns, attributes) in the data set. The field names must be the keys in this dictionary.

- **fields_default**
  Default string if a field is not found (for writing records into a sequential data set). The default value is an empty string.
• **missing_values**
  A list of strings that correspond to missing values in a data set. This list is used in the calculation of matching weights in the record linkage process, i.e. if a value in a record is equal to a missing value as defined in this `missing_values` list, the corresponding matching weight is set to a missing value weight (see Section 9.2 for more details).

Sequential data sets (i.e. COL, CSV and SQL) have the following additional attribute that can be set when a data set is initialised.

• **strip_fields**
  A flag that can be set to `True` or `False`. If set to `True` (the default value), all fields are stripped of their whitespaces before they are written into a data set or read from a data set. If set to `False`, whitespaces are not stripped off field values.

Two attributes that should be read only (one of them for sequential access data sets only) are

• **num_records**
  The total number of records in a data set. Note that the value of this attribute will be 0 for sequential access data sets if initialised in write access mode. For all other access modes, the number of stored records in the data set is calculated, which might take a while (depending on the data set implementation).

• **next_record_num**
  This attribute is only available in sequential access data sets. It is the record number of the next record to be accessed. When a data set is initialised, the value of this attribute will be 0, and after having read one record it will be 1. If a block of records is read, it will be the number of the next record in the data set.

Besides initialisation, all data set implementations have the following methods to read or write records.

• **read_record()** *(sequential access data sets)*
  **read_record(record_number)** *(direct random access data sets)*
  For reading one single record. For sequential data sets, the record with the `next_record_num` number is returned (and the value of `next_record_num` increased by one). If no more records are available, None is returned.
  For direct random access a record number must be given to this method and the record with the corresponding `_rec_num_` is returned. If the record with the given number is not in the data set, None is returned and a warning message is triggered.

• **read_records(first_record, number_records)** *(sequential access data sets)*
  **read_records(record_number_list)** *(direct random access data sets)*
  For sequential access data sets, a block of records can be read using this method, starting from the record with number `first_record` to (including) `first_record+number_records-1`. After a block of records is read successfully, the value of `next_record_num` is set to the number of the record following the block. Records are returned in a list.
  For direct random access data sets, a list of record numbers must be given to this method, and the corresponding records are returned in a list. The list of returned records can be shorter than the list of record numbers if a record in the list is not in the data set and thus can not be read.

• **write_record(record)**
  Write one record into a data set. For sequential access data sets, the record is written to the end of the data set (so the hidden field `_rec_num_` is not used), while for direct random access data sets the record written with the value of `_rec_num_` as identifier (if there was already a record with this number in the data set it is overwritten). For sequential data sets, the value of `num_records` is increased by one in any case, while for direct random access data sets it is only increased if previously no record was stored in the data sets with the given record number.
• **write_records**(record_list)
  
  Write the list of records into the data set. For sequential access data sets, the records are written to the end of
  the data set, and the value of num_records is increased by the number of records written, while for direct
  random access data sets the hidden fields _rec_num_ are used to write records into the data set. The value of
  num_records with direct random access data sets is only increased by the number of records with new record
  numbers.

• **finalise()**
  
  With this routine the access to a data set can be finalised, i.e. depending on the implementation a connection to
  the database is closed or an underlying file is closed. After a call to this routine no data set access is possible,
  but the data set can be re-initialised with a new (or the same) access mode. Note that for a memory based data
  set all data is lost when it is finalised.

Direct random access data sets have another method that allows them to be re-initialise.

• **re Initialise**(access_mode) *(direct random access data sets)*
  
  This method can be used to finalise and re-initialise a random access data set, which is needed for parallel runs
  of Febrl where for example only the host process writes standardised records into a temporary data set in the
  cleaning and standardisation phase (see note below), but all processes need to be able to read these cleaned
  records in the linkage or deduplication phase. So, a re-initialisation can be used to synchronise the random data
  sets for all processes (which for a memory based data set means it has to be copied to all processes).
  This method can be called with or without a new access mode. If no access_mode argument is given, the
  data set will be initialised in the same access mode. Alternatively a new access mode can be given.
  Depending on the setting for parallel write access (see note below), random access data sets should be re-
  initialised in read access mode so that all processes are able to read all the records in such a data set.

**Note:** When Febrl is run in parallel, two different forms of write data set access are possible. First, only the host
process (where Febrl has been started on) is writing into data sets, or secondly, all processes are writing into (local)
data sets. When initialising a Febrl project object (as described in Chapter 5), it is possible to configure this parallel
write access mode by setting the argument parallel_write to either 'host' (the default) or 'all'. In the
  case where all processes are writing, each of them will write into its own (local) data sets. This is done by adding the
  process number (for example P0 for the host, P1, P2, etc. for the other processes) to the file names. For the SQL data
  set only the 'host' write access mode is currently supported.

In the following sections details on the data set implementations and their specific attributes are given.

### 12.1 COL Data Set Implementation

Text files with fields with fixed column width are commonly used. Fields are normally specified with the start column
and the field width, or with a start and an end column. Febrl uses the start column (starting with zero) and field width
(or length) format to define fields. The file extension of such files is often '.txt'. This data set implementation
allows sequential access only.

The fields attribute of a COL data set must be a dictionary where the keys are the field names and the values are
tuples with start column (starting from 0) and field width.

Additional attributes (besides the general data set attributes as described above) for a COL data set are

• **file_name**
  
  A string containing the name of the underlying COL text file.

• **header_lines**
  
  The number of header lines at the beginning of the COL file that have to be skipped in read access mode. The
default value is 0, which corresponds to no header line.
- write_header
  A flag that can be set to True or False. If set to True, a header line with the field names is written at the beginning of the COL file if the data set is initialised in write mode or in append mode (if the file is empty). The default value is False, i.e. no header line will be written. Note that field names will be truncated or expanded with white spaces in order to fit into the field formats (number of columns defined for the fields).

Note: When a COL data set is initialised in read access mode, it is possible to define fields that are overlapping, and gaps between fields are also possible. For example

```python
def fields = {'hospitalcode':(10,4),
    'year':(14,8),
    'yearhospocode':(10,12),
    'name':(30,20),
    'address':(60,50)}
```

However, for COL data sets initialised in write or append mode, the field definitions must not be overlapping nor must there be gaps between field definitions.

Note: In its current implementation, a COL data set can only consist of one underlying COL text file. The handling of multiple files as one data set will be implemented in a future version of Febrl.

The following example shows how to initialise a COL data set and how to access it in read mode. It is assumed that the dataset.py module has been imported using the import dataset command.

```python
# ====================================================================
mydata = dataset.DataSetCOL(name = 'hospital-data',
    description = 'Hospital data from 1990-2000',
    access_right = 'read',
    header_lines = 1,
    file_name = './data/hospital.txt',
    fields = {'year':(0,4),
        'surname':(4,10),
        'givenname':(14,10),
        'dob':(24,8),
        'address':(32,30),
        'postcode':(62,4),
        'state':(66,3)},
    fields_default = '',
    strip_fields = True,
    missing_values = ['','missing'])
print mydata.num_records # Print total number for records
first_record = mydata.read_record() # Returns one record
hundred_records = mydata.read_records(0,100) # Read 100 records
ten_records = mydata.read_records(2000,10) # Read another 10 records
mydata.finalise() # Close file, finalise access to data set
```

12.1. COL Data Set Implementation
Text files with comma separated values are common, as they are a portable way to store data from spreadsheets or database tables. Often such files have the file extension '.csv'. This data set implementation allows sequential access only.

The `fields` attribute of a CSV data set must be a dictionary where the keys are the field names and the values are the corresponding column numbers (starting with 0).

Additional attributes (besides the general data set attributes as described above) for a CSV data set are

- `file_name`
  A string containing the name of the underlying CSV text file.

- `header_lines`
  The number of header lines at the beginning of the CSV file that have to be skipped in read access mode. The default value is 0, which corresponds to no header line.

- `write_header`
  A flag that can be set to True or False. If set to True, a header line with the field names is written at the beginning of the CSV file if the data set is initialised in write mode or in append mode (if the file is empty). The default value is False, i.e. no header line will be written.

- `write_quote_char`
  It is possible to quote each field in the CSV file when records are written to the file. The default value of the quote character is an empty string, which means the fields are not quoted. A common quote character is " (double quotes).

The following example shows how to initialise a CSV data set and how to access it in read mode. It is assumed that the `dataset.py` module has been imported using the `import dataset` command.

```python
# ====================================================================
mydata = dataset.DataSetCSV(name = 'hospital-data',
                            description = 'Hospital data from 1990-2000',
                            access_right = 'read',
                            header_lines = 1,
                            file_name = './data/hospital.csv',
                            fields = {'year':0,
                                      'surname':1,
                                      'givenname':2,
                                      'dob':12,
                                      'address':7,
                                      'postcode':8,
                                      'state':9},
                            fields_default = '',
                            strip_fields = True,
                            missing_values = ['', 'missing'])

print mydata.num_records  # Print total number for records
first_record = mydata.read_record()  # Returns one record
hundred_records = mydata.read_records(1000,10)  # Read 100 records
ten_records = mydata.read_records(2000,10)  # Read another 10 records
mydata.finalise()  # Close file, finalise access to data set
```

90 Chapter 12. Data Set Access
Note: In its current implementation, a CSV data set can only consist of one underlying CSV text file. The handling of multiple files as one data set will be implemented in a future version of Febrl.

12.3 SQL Data Set Implementation

This data set implementation allows access to the popular MySQL open source database system. It uses the Python module MySQLdb which has to be installed separately.\(^2\)

Note: Support for a wider range of SQL databases will be added to Febrl in a future release. More information on the Python database API can be found at [http://www.python.org/topics/database/](http://www.python.org/topics/database/), including a list of Python modules for various (commercial and open source) databases.

The SQL data set implementation currently allows sequential access only.

The fields attribute of an SQL data set must be a dictionary where the keys are the field names and the values are the names of the corresponding table columns in the SQL database.

If an SQL data set is initialised in write or append mode it is assumed that the table is already created and contains the necessary columns. The main reason for this is when creating an SQL table it is necessary to give the types and length of all table columns, information which are not available when a data set is initialised.

Additional attributes (besides the general data set attributes as described above) for a SQL data set are

- **table_name**
  A string containing the name of the underlying SQL table.

- **database_name**
  The name of the database that contains the table to be accessed.

- **database_user**
  The name of the database user.

- **database_password**
  The corresponding database password for the given user. If this password is not set when the data set is initialised, it will be queried interactively when the connection to the database is established.

- **database_block_size**
  For efficiency reasons reading records from a database table is done in blocks. The default number of records in a block is set to 10,000, but this value can be adjusted with this attributed.

The following example shows how to initialise a SQL data set and how to access it in read mode. It is assumed that the dataset.py module has been imported using the import dataset command.

\(^2\) Available for download from [http://sourceforge.net/projects/mysql-python](http://sourceforge.net/projects/mysql-python)
mydata = dataset.DataSetSQL(name = 'hospital-data',
description = 'Hospital data from 1990-2000',
access_right = 'read',
database_name = 'hospital-data',
database_user = 'exampleuser',
table_name = 'hospital',
fields = {'year': 'year',
          'surname': 'sname',
          'givenname': 'gname',
          'dob': 'dateofbirth',
          'address': 'wayfare',
          'postcode': 'pcode',
          'state': 'territory'},
fields_default = '',
strip_fields = True,
missing_values = ['', 'missing'])

print mydata.num_records # Print total number for records

first_record = mydata.read_record() # Returns one record

hundred_records = mydata.read_records(1000, 100) # Read 100 records
ten_records = mydata.read_records(2000, 10) # Read another 10 records

mydata.finalise() # Close file, finalise access to data set

Note: In its current implementation, an SQL data set can only consist of one underlying database table. The handling of multiple tables as one data set will be implemented in a future version of Febrl.

12.4 Shelve Data Set Implementation

The shelve data set uses the Python standard module shelve.py, which provides a file-based hash table (dictionary) that allows efficient storage and access of arbitrary records. Thus, a shelve data set becomes an efficient and convenient data set implementation for temporary persistent storage of records. This data set implementation is for direct random access only.

Note: The Python shelve module is based on a database like dbm, gdbm or bsddb. Unfortunately, these database libraries seem to be badly broken when used within Python 2.2 (i.e. they crash when trying to load several thousand records into a shelve). Starting with Febrl version 0.2.1 we are therefor supporting the use of the external module bsddb3 which can be downloaded from http://pybsddb.sourceforge.net. The Berkeley database library itself is available under an open source software license from Sleepycat Software at http://www.sleepycat.com. The shelve data set implementation automatically detects if the bsddb3 module is installed or not, and will use it if it is available.

The fields attribute of a shelve data set must be a dictionary where the keys are the field names. The corresponding values are not used and can thus be anything, e.g. an empty string or an integer number. They are not needed to access records or the fields within records. Two additional attributes (besides the general data set attributes as described above) for a shelve data set are
- **file_name**
  A string containing the name of the underlying database file (or files). Don’t use a file name with extension, as for certain underlying database implementations more than one file with different file extensions will be created.

- **clear**
  A flag that can be set to True or False. If set to True, the content of the database will be cleared when it is opened in write or readwrite access modes. Otherwise, the database is kept. In read access mode this attribute has no functionality (clearing a database in read only access would mean to have no records to read).

The following example shows how to initialise a shelve data set and how to access it in read/write mode. It is assumed that the dataset.py module has been imported using the import dataset command.

```python
# ====================================================================
mydata = dataset.DataSetShelve(name = 'hospital-data',
description = 'Hospital data from 1990-2000',
access_right = 'readwrite',
file_name = 'hospital',
clear = True,
fields = {
    'year': '',
    'surname': '',
    'givenname': '',
    'dob': '',
    'address': '',
    'postcode': '',
    'state': ''},
fields_default = '',
missing_values = ['', 'missing'])

print mydata.num_records  # Print total number for records

first_record = {'surname': 'miller', 'givenname': 'peter', 'state': 'act',
                '_rec_num_': 0}

mydata.write_record(first_record)

more_records = [{'surname': 'smith', 'givenname': 'dave', 'dob': '1966',
                  '_rec_num_': 1},
                  {'surname': 'winkler', 'givenname': 'harry',
                   '_rec_num_': 42},
                  {'surname': 'paul', 'postcode': '2100', 'state': 'nsw',
                   '_rec_num_': 0}]

mydata.write_records(more_records)

print mydata.num_records  # Print total number for records (3)

record = mydata.read_record(42)

record_list = mydata.read_records([0, 1])

mydata.re_initialise()  # Re-initialise data set in readwrite mode

record = mydata.read_record(1)

mydata.write_record(first_record)
```
mydata.re_initialise('read')  # Re-initialise data set in read mode

record_list2 = mydata.read_records([42,0,1])

mydata.finalise()  # Close file, finalise access to data set

Note that record numbers (values in the hidden field _rec_num_) do not necessarily need to be in a consecutive range, as shown in the example above. Also, records can be overwritten at any time if a new record with an already existing record number value is written to the data set.

12.5 Memory Data Set Implementation

The memory data set uses the Python dictionary data structure – which are basically hash tables – to temporarily store records in main memory. This data set allows direct random access. This is an efficient temporary storage for records that do not need to be made persistent, e.g. for cleaned and standardised records before they are linked with another data set.

The only possible access mode for this data set implementation is of course ‘readwrite’ (as the ‘read’ or ‘write’ only access modes would prohibit the use of memory based data sets).

The fields attribute of a memory data set must be a dictionary where the keys are the field names. The corresponding values are not used and can thus be anything, e.g. an empty string or an integer number. They are not needed to access records or the fields within records.

Note that all records are lost when a memory data set is finalised or a program finishes. Use other data sets for persistent storage.

Besides the general data set attributes as described above no additional attribute (like file name) is needed.

The following example shows how to initialise a memory data set and how to access it in read/write mode. It is assumed that the dataset.py module has been imported using the import dataset command.

```python
# mydata = dataset.DataSetMemory(name = 'hospital-tmp-data',
description = 'Hospital data from 1990-2000',
access_right = 'readwrite',
file_name = 'hospital',
fields = {'year': '',
'surname': '',
'givenname': '',
'dob': '',
'address': '',
'postcode': '',
'state': ''},
fields_default = '',
strip_fields = True,
missing_values = ['','missing'])

print mydata.num_records  # Print total number for records (0)

first_record = {'surname': 'miller', 'givenname': 'peter', 'state': 'act',
'_rec_num_': 0}
```
mydata.write_record(first_record)

more_records = [{'surname':'smith','givenname':'dave','dob':'1966',
                '_rec_num_':1},
                {'surname':'winkler','givenname':'harry',
                 '_rec_num_':42},
                {'surname':'paul','postcode':'2100','state':'nsw',
                 '_rec_num_':0}]

mydata.write_records(more_records)

print mydata.num_records  # Print total number for records (3)

record = mydata.read_record(42)

record_list = mydata.read_records([0,1])

mydata.finalise()  # Finalise access to data set, all records are lost

Note that record numbers (values in the hidden field \_rec_num\_) not necessarily need to be in a consecutive range, as shown in the example above. Also, records can be overwritten at any time if a new record with an already existing record number value is written to the data set.
Look-up and Frequency Tables

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 created and edited by the user. Four different types of look-up tables and corresponding file formats as used in the Febrl system are described in this chapter. Their access routines are implemented in the module `lookup.py`.

- **Correction list**
  These lists contain strings (characters or words) and their replacements. They are used in the initial cleaning step in the data standardisation process.

- **Tagging look-up table**
  Similar to correction lists, these tables contain strings and their replacement. Additionally, groups of table entries are assigned a tag, which is then used in the tagging step within the data standardisation process. Tagging look-up tables are mainly used for names and addresses.

- **Frequency look-up table**
  These tables contain words and integer numbers that correspond to the frequency of the word as listed in an external frequency file. Such frequency tables are used within the record linkage process to calculate frequency dependent matching weights.

- **Geographic location look-up table**
  These tables contain words and their corresponding geographical location given as a numerical longitude and latitude pair. For example, postcode locations can be used to calculate the distance between two postcodes, which can then be used in a field comparison function to calculate a matching weight.

For each of these four different tables or lists the corresponding file format is described in the following sections. Common to all file formats 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 look-up table or correction list entry.

For the tagging, frequency and geographic location look-up tables, it is possible to load more than one file into one combined look-up table (see the examples given in the following sections).

Also for the tagging, frequency and geographic location look-up tables an additional argument that can be given when a table is initialised is `default`, a string that is returned when the table is queried for a value that is not stored in the table.

---

\(^1\) The AutoMatch as formerly sold by MatchWare Technologies derived quite small frequency weight tables directly from the input files. The Febrl system allows 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.
13.1 Correction List

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 should have a file extension `.lst`. The format of these files is as follows:

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

  \[ \text{replacement} := \text{values} \]

  where \text{values} is a list of one or more strings separated by commas. Each \text{value} in this list is replaced with the \text{replacement} character or string on the left hand-side of the entry.

- All \text{replacement} and \text{value} strings have to be quoted, either with single or double quotes.

- An entry can be longer than one line, in which case the second and following lines consist of the \text{values} list only.

- Comment lines are lines that start with a `#` character.

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

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

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

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

In the above example, 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 \text{value} and \text{replacement} pairs is internally sorted with decreasing length of the \text{values}. Long \text{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 `. `. Assuming the `lookup.py` module has been imported using the `import lookup` command, an example correction list can be initialised and loaded as shown in the following example.
# address_correction_list = lookup.CorrectionList(name = 'AddrCorrList')
# address_correction_list.load('address_corr_list.lst')
# print address_correction_list.length  # Number of entries in the list

## 13.2 Tagging Look-up Table

A tagging look-up table file contains one or more blocks of entries, with all entries in a block are being assigned the same tag. Tagging look-up table files should 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 strings (not quoted) separated by commas, 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 `key` and a `value` can consist of more than one word, separated by spaces.

- 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 triggered and the program stops. The user then manually has to correct this conflict in the file.

- Comment lines are lines that start with a `#` character.

The following examples are extracted from the `name.misc.tbl` and `territory.tbl` files.
It is possible to load more than one tagging look-up table file into one combined tagging look-up table, by simply giving a list of file names when the table is loaded, as shown in the example below. If an entry is listed in different files with different tags, and error is triggered.

The default value for the attribute default is an empty string '', i.e. if a value is looked up in a table that does not exist, an empty string is returned. The default value can be changed when a tagging look-up table is initialised using the default argument as shown in the example below.

After one or more tagging files have been loaded into a tagging look-up table, the attribute max_key_length is set to the maximal length in words of all keys in the look-up table. If for example the longest key in a look-up table is 'south west rocks' then the value of max_key_length would be 3.

Assuming the lookup.py module has been imported using the import lookup command, an example tagging look-up table can be initialised and loaded from several files as shown in the following example.
name_tagging_table = lookup.TagLookupTable(name = 'NameTagTable',
    default = 'missing')

name_tagging_table.load(['./data/givenname_f.tbl',
    './data/givenname_m.tbl',
    './data/name_prefix.tbl',
    './data/name_misc.tbl',
    './data/saints.tbl',
    './data/surname.tbl',
    './data/title.tbl'])

print name_tagging_table.length
print name_tagging_table.max_key_length
print name_tagging_table[('peter',)] # Prints: ('peter', 'GM')
print name_tagging_table['xyg0542w'] # Assume not in table, 'missing'
    # will be returned

13.3 Frequency Look-up Table

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, thus they are simple CSV (comma separated values) files, as for example created by a spreadsheet. The first column contains words and the second column contains the corresponding frequency counts (positive integer numbers). These files should have a file extension '.csv'. The following example is extracted from a surname frequency look-up table.

```
# ====================================================================

dijkstra,3
miller,4325
smith,22540
```

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.

It is possible to load more than one frequency look-up table files into one combined frequency look-up table, by simply giving a list of file names when the table is loaded, as shown in the example below. If an entry is listed in more than one file, its frequency counts are simply added up.

The default value for the attribute default is the value 1, i.e. if a value is searched in a table that does not exist, the default value 1 is returned. The default value can be changed when a frequency look-up table is initialised using the default argument as shown in the example below.

After a frequency look-up table has been loaded from one or more files, the total sum of all frequency counts is stored in the attribute sum.

Assuming the lookup.py module has been imported using the import lookup command, an example frequency look-up table can be initialised and loaded from several files as shown in the following example.
name_freq_table = lookup.FrequencyLookupTable(name = 'NameFreqTable')
name_freq_table.load(['surname_english.csv','surname_french.csv'])
print name_freq_table.sum
print name_freq_table.length
print name_freq_table['miller']  # Returns for example 246
print name_freq_table['leroc']  # Returns for example 42
print name_freq_table['deutschmann']  # Should return default value 1
# assuming it’s not in the table

13.4 Geographic Location Look-up Table

The last type of look-up table files are lists of words with their corresponding geographic locations as numerical longitude and latitude value pairs. These files contain three columns separated by a comma, thus they are simple CSV (comma separated values) files, similar to frequency look-up table files. The first column contains words, the second column contains the longitude values and the third column contains the latitude values. Longitude values need to be between -180.0 and 180.0 and latitude values between -90.0 and 90.0. These files should have a file extension '.csv'. The following example is extracted from a postcode geographic location look-up table.

# ================================================================
2000, 151.20710, -33.87060
2007, 151.19761, -33.87849
2008, 151.19655, -33.88829
2009, 151.19395, -33.87014
2010, 151.21524, -33.88299

It is possible to load more than one geographic location look-up table files into one combined geographic location look-up table, by simply giving a list of file names when the table is loaded, as shown in the example below. If an entry is listed in more than one file with different locations, an error is triggered.

The default value for the attribute default is the value [], i.e. an empty list. So if a value is searched in a table that does not exist, an empty list is returned and the calling program can act upon this. The default value can be changed when a geographic location look-up table is initialised using the default argument.

Assuming the lookup.py module has been imported using the import lookup command, an example geographic location look-up table can be initialised and loaded from several files as shown in the following example.
pc_location_table =lookup.GeocodeLookupTable(name = 'PCGeoLocTable')

pc_location_table.load(['postcode_nsw.csv',
    'postcode_qld.csv',
    'postcode_tas.csv',
    'postcode_act.csv'])

print pc_location_table.length

print pc_location_table['2000']  # Returns [151.2071, -33.8706]
print pc_location_table['9999']  # Should return [] because '9999' is not a valid postcode
Logging and Verbose Output

The Febrl system is provided with a logging system and it can display and save to a file logging information with different levels (i.e. different volumes of output). A project log can be defined in a project.py module as shown in the code example below. It is assumed that the febrl.py module has been imported using the import febrl command (as the ProjectLog class is implemented in the febrl module), and a project myproject has been initialised (see also Chapter 5).

```python
# ====================================================================
mylog = febrl.ProjectLog(file_name = 'myfebrl.log',
                         project = myproject,
                         log_level = 1,
                         verbose_level = 1,
                         clear_log = True,
                         no_warn = False,
                         parallel_print = 'host')
```

The following arguments can be given to a project log when it is initialised.

- **file_name**
  The name of the text file logging information is saved into. If set to None no logging information will be saved (but still displayed).

- **project**
  A reference to a project.

- **log_level**
  An integer number in the range of 0 to 3. Zero means no information is logged into the log file (if specified), and numbers between 1 and 3 mean an increasing of the logging information. When set to level 1, only overview messages are saved, while with level 3 very detailed (at record level) messages are saved.

- **verbose_level**
  Similar to log_level this argument controls the level of information that is displayed on standard output (i.e. the terminal where Febrl is running). Valid are numbers in the range of 0 to 3, with nothing being displayed with level 0, and lost of information displayed at level 3.

- **parallel_print**
  This argument defines how displaying and logging is done when Febrl is run in parallel. Possible values are either ’host’ (the default) in which case only the host process (where Febrl has been started on) is displaying and logging messages, or ’all’ in which case all processes are displaying and logging. Note that currently both error and warning messages are displayed by all processes.
• **clear_log**
  A flag that can be set to True or False (default is False). When set to True, all information stored in the given log file is first erased, and a new log is started. Otherwise, logging information is appended to a log file (if it exists, otherwise it is created first).

• **no_warnings**
  A flag that can be set to True or False (default is False). If set to True no warning messages are logged or displayed (independent of the log and verbose level), while if set to False all warning messages are logged and displayed (again independent of the log and verbose levels).

**Note:** The current version of the *Febrl* system only supports logging into text files. Future version will provide logging into other formats as well, for example HTML, XML or \LaTeX.
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 your `.cshrc` or `.bashrc` file by adding the path to your Python installation.

See Appendix D for a list of all files distributed with the current version of Febrl.

The Febrl system is configured and controlled by a project module derived from either the supplied `project-linkage.py`, `project-deduplicate.py` or `project-standardise.py` modules. See Chapter 5 for more information on these modules.

For hidden Markov model training, the two programs `tagdata.py` (for tagging training records) and `trainhmm.py` (for training a hidden Markov module) need to be used. Hidden Markov model training and the two mentioned programs are described in detail in Chapter 8. The auxiliary program `randomselect.py` can be used to randomly select a sub-set of a data set (see Section 11.1 for more details).

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-0.2.2.tar.gz
```

or similar.

After unzipping or untarring the Febrl distribution, change to the Febrl directory and run the various unit test programs supplied with Febrl from there. Unit test programs are file with names of the form `modulenameTest.py`. You can easily run these unit tests by typing for example (to test the string encoding module `encode.py`)

```
python encodeTest.py
```

All unit test programs should run without any errors. If you do get error message with these test programs, please send an e-mail to the Febrl authors including the error message.

Next, according to your needs make a copy of the `project-linkage.py`, `project-deduplicate.py` or `project-standardise.py` module and modify this copy according to your data set(s). That’s it at this stage. To
run a project module, for example your `myproject.py`, simply type

```
python myproject.py
```

In order to be able to run **Febrl** in parallel you need to have **MPI** and **Pypar** installed on your system. Please see Chapter 16 for more details on parallelism.

**Note:** In order to be able to use disk based temporary data sets (i.e. *Shelve* data sets as discussed in Section 12.4) you should install the Berkeley database library as discussed in Section 12.4 because the standard `shelve` module supplied with Python 2.2 does crash when large number of records are written into such a data set.

**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 and will allow many people to share a single installation of **Febrl** on a single shared computer.
Parallelism is built into the current version of Febrl transparent to the user. Running Febrl in parallel allows to solve problems with a shorter run-time compared to run them sequentially, or alternatively allows to solve larger problems due to the (usual) availability of larger amounts of memory on parallel computing platforms.

In order to be able to use the parallel functionality of Febrl the following software must be installed on your computing platform (assuming a parallel hardware like a multiprocessor or a cluster of personal computers or workstations is available).

- **MPI** (Message Passing Interface)
  MPI is a quasi standard for parallel programming on distributed memory platforms. It defines a large number of routines for communicating data (i.e. messages) between processors. While MPI itself only defines the standard (so that programs written in MPI are portable to various parallel platforms), there are different implementations available, some from vendors of parallel (super-) computers, others as free downloadable packages. Please see the MPI web page at
  http://www-unix.mcs.anl.gov/mpi/
  for more information on MPI and links to various implementations. Note that on many platforms administrator (or superuser) access rights are needed in order to be able to install an MPI implementation.

- **Pypar**
  Pypar is an efficient and easy-to-use module that allows programs/scripts written in the Python programming language to run in parallel on multiple processors and communicate using MPI. See the Pypar web page at
  http://datamining.anu.edu.au/pypar
  for more information and to download the package.

Once both MPI and Pypar are installed and tested successfully, you can run Febrl in parallel by using the mpirun command of your MPI implementation. For example, if you have a Febrl project module called myproject.py and you have a parallel platform with 8 processors, you can run Febrl in parallel by using

    mpirun -np 8 python myproject.py

**Note:** In order to be able to access the data sets and look-up tables, all processors must be able to have access to the directory (and sub-directories) defined in the Febrl object attribute febrl_path. Future version of Febrl will allow a much more sophisticated definition of parallelisation settings.

**Note:** Note that parallelism within Febrl is in its initial stages, and we would like to ask people who are interested in this area to contact the authors for further exchange of detailed information and experiences.
Warning: While doing extensive tests running parallel Febrl jobs in some cases we got slightly different numerical results when linking or deduplication larger data sets (with around 100,000 or more records). When comparing the results from running the same job on different numbers of processes, the final weights (as stored in a results file) sometimes differ in the range of $10^{-4}$ (fourth or fifth digit after the comma).

So far we have not found the cause of these problems, which might be part of our local MPI/Pypar installation, or within one of the Febrl modules.

We are currently working on this problem and will publish an updated and hopefully correct version of Febrl as soon as the problem is solved.
Hidden Markov Model States

The following two tables contain all possible states for the name and address hidden Markov models, respectively, as discussed in Chapters 7 and 8.

Table A.1: States for Name HMM

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>titl</td>
<td>Title state</td>
</tr>
<tr>
<td>baby</td>
<td>State for <em>baby of, son of or daughter of</em></td>
</tr>
<tr>
<td>knwn</td>
<td>State for <em>known as</em></td>
</tr>
<tr>
<td>andor</td>
<td>State for <em>and or</em></td>
</tr>
<tr>
<td>gname1</td>
<td>Given name state 1</td>
</tr>
<tr>
<td>gname2</td>
<td>Given name state 2</td>
</tr>
<tr>
<td>ghyph</td>
<td>Given name hyphen state</td>
</tr>
<tr>
<td>gopbr</td>
<td>Given name opening bracket state</td>
</tr>
<tr>
<td>gcbr</td>
<td>Given name closing bracket state</td>
</tr>
<tr>
<td>agname1</td>
<td>Alternative given name state 1</td>
</tr>
<tr>
<td>agname2</td>
<td>Alternative given name state 2</td>
</tr>
<tr>
<td>coma</td>
<td>State for comma</td>
</tr>
<tr>
<td>sname1</td>
<td>Surname state 1</td>
</tr>
<tr>
<td>sname2</td>
<td>Surname state 2</td>
</tr>
<tr>
<td>shyph</td>
<td>Surname hyphen state</td>
</tr>
<tr>
<td>sopbr</td>
<td>Surname opening bracket state</td>
</tr>
<tr>
<td>sclbr</td>
<td>Surname closing bracket state</td>
</tr>
<tr>
<td>asname1</td>
<td>Alternative surname state 1</td>
</tr>
<tr>
<td>asname2</td>
<td>Alternative surname state 2</td>
</tr>
<tr>
<td>pref1</td>
<td>Name prefix state 1</td>
</tr>
<tr>
<td>pref2</td>
<td>Name prefix state 2</td>
</tr>
<tr>
<td>rubb</td>
<td>Rubbish state, for elements to be thrown away</td>
</tr>
</tbody>
</table>
Table A.2: States for Address HMM

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wfnu</td>
<td>Wayfare number state</td>
</tr>
<tr>
<td>wfn1</td>
<td>Wayfare name state 1</td>
</tr>
<tr>
<td>wfn2</td>
<td>Wayfare name state 2</td>
</tr>
<tr>
<td>wfql</td>
<td>Wayfare qualifier state</td>
</tr>
<tr>
<td>wfty</td>
<td>Wayfare type state</td>
</tr>
<tr>
<td>unnu</td>
<td>Unit number state</td>
</tr>
<tr>
<td>unty</td>
<td>Unit type state</td>
</tr>
<tr>
<td>prna1</td>
<td>Property name state 1</td>
</tr>
<tr>
<td>prna2</td>
<td>Property name state 2</td>
</tr>
<tr>
<td>inna1</td>
<td>Institution name state 1</td>
</tr>
<tr>
<td>inna2</td>
<td>Institution name state 2</td>
</tr>
<tr>
<td>inty</td>
<td>Institution type state</td>
</tr>
<tr>
<td>panu</td>
<td>Postal address number state</td>
</tr>
<tr>
<td>paty</td>
<td>Postal address type state</td>
</tr>
<tr>
<td>hyph</td>
<td>State for hyphen</td>
</tr>
<tr>
<td>sla</td>
<td>State for slash</td>
</tr>
<tr>
<td>coma</td>
<td>State for comma</td>
</tr>
<tr>
<td>opbr</td>
<td>Opening bracket state</td>
</tr>
<tr>
<td>clbr</td>
<td>Closing bracket state</td>
</tr>
<tr>
<td>loc1</td>
<td>Locality name state 1</td>
</tr>
<tr>
<td>loc2</td>
<td>Locality name state 2</td>
</tr>
<tr>
<td>locql</td>
<td>Locality qualifier state</td>
</tr>
<tr>
<td>pc</td>
<td>Postcode state</td>
</tr>
<tr>
<td>ter1</td>
<td>Territory name state 1</td>
</tr>
<tr>
<td>ter2</td>
<td>Territory name state 2</td>
</tr>
<tr>
<td>cntr1</td>
<td>Country name state 1</td>
</tr>
<tr>
<td>cntr2</td>
<td>Country name state 2</td>
</tr>
<tr>
<td>rubb</td>
<td>Rubbish state, for elements to be thrown away</td>
</tr>
</tbody>
</table>
List of Tags

The following table contains all possible tags for the name and address component, respectively, as discussed in Chapters 7 and 8.

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

This section describes the rule-based name standardisation algorithms as implemented in the routine `get_names_rules()` in the module `name.py`.

C.1 Input

The required inputs for this routine are a list of words and a list of tags as created in the data cleaning and tagging steps (see Chapter 6 for more details). Additionally, a third parameter `first_name_comp` is needed that gives the system a hint on which name component (given- or surname) is most likely to be at the beginning of the input word list. For example, after cleaning and tagging, the input name string

`'del (van a.k.a. peter) miller, phd'`

is converted into the following word and tag lists

```plaintext
['del', '|', 'van', 'known as', 'peter', '|', 'miller', ',']
['PR', 'VB', 'PR', 'SP', 'GM', 'VB', 'UN', 'CO']
```

The first name component argument would be set to `'gname'` in this example, because this name starts with a given name followed by the surname. Note that all title words (in this example `'phd'`) are processed and removed from the input word and tag lists before the rule based name segmentation is started.

C.2 Process Overview

In a first step, the algorithm goes through the input word and tag lists and allocates them into one of five sub-lists as explained in more details below. These sub-lists are an indication of what the elements in them are likely to be. In a second step the sub-lists are checked according to various rules to decide into which category (given name, alternative given name, surname or alternative surname) each word belongs.

C.2.1 Step 1: Allocating the elements into one of the five sub-lists

The following are the five sub-lists.

0: Words that are likely to be given names. All words/tags until the first separator (e.g. `'known as'`).
1: Words that are likely to be alternative given names. All words/tags in the first alternative given word area.
2: Words that are likely to be surnames. All words/tags until the next separator or the end of the list.
3: Words that are likely to be alternative surnames. All words/tags in the second alternative given word area or until the end of the list.

4: Words that are likely to be second alternative surname or until the end of the list. All words/tags after second alternative word area or the end of the list.

Pseudo code

The following is a pseudo code for step 1.

```
if no separator or vertical bar in the input,
    then all elements in word-list belong to sub-list 0.
    step 1 finishes.
pointer points to sub-list 0.
for each element is in word-list:
    if the element is a separator:
        store the next element(s) to the pointed sub-list.
        move the pointer to the next sub-list.
    if the element is a vertical bar:
        move the pointer to the next sub-list.
        store all elements until the next vertical bar in the pointed sub-list.
        if the word after the next vertical bar is not a separator
            move the pointer to the next sub-list.
    if the element is anything else:
        store the element in the pointed sub-list.
```

Output

The output of the first step is the five sub-lists containing words and the corresponding tags. Note that not all sub-lists do have to contain words, in fact in most cases (general simple names without any alternative names) the complete name is stored in sub-list 0.

The following is the output (words and tags) from the first step using the example input shown above.

0 (potential given names): ['del'] ['PR']
1 (potential alternative given names): ['van', 'peter'] ['PR', 'GM']
2 (potential surnames): ['miller', ','] ['UN', 'CO']
3 (potential alternative surnames): []
4 (potential second alternative surnames): []

C.2.2 Step 2: Parse each sub-list and assign into appropriate output name component

During the second step, the system goes through each sub-list created in step 1, starting from sub-list 4 backwards. According to various hard-coded rules, it analyses the tags in a sub-list and then allocates the corresponding words into one of the following output name components (lists of words):
Given names

Alternative given names

Surnames

Alternative surnames

Pseudo code

The following is the pseudo code for the second step.

```plaintext
if sub-list 4 is not empty:
  if givenname is usually first, store its elements as alternative surname.
  if surname is usually first, store its elements as alternative givenname.
if sub-list 3 is not empty:
  if givenname is usually first, append its elements as alternative surname.
  if surname is usually first, append its elements as alternative givenname.
if sub-list 2 is not empty:
  if givenname is usually first, store surname element as surname, and
      append alternative surname element to alternative surname.
  if surname is usually first, store givenname element as givenname, and
      append alternative givenname element to alternative givenname.
if sub-list 1 is not empty:
  if the first name component (e.g. givenname) doesn’t have an alternative
      name, but the second name component (e.g. surname) does, then store the
      elements alternative surname.
  if the first name component has an alternative name, then sub-list 2
      was not empty, then store the elements as either alternative surname (if surname
      is usually first) or alternative givenname (if given name is usually first).
sub-list 0 should be non-empty all the time:
  if both sub-list 1 and 2 are not empty, then the elements in this list is
    either surname & alternative surname (if surname is usually first) , or
    givenname & alternative givenname (if given name is usually first).
  if sub-list 2 is empty, then for both cases sub-list 1 empty or not
    this sub-list 0 contains surname and givenname.
```

The following pseudo code shows the rules if only sub-list 0 is non-empty (i.e. no other sub-list contains any words, the case with all simple names).

```plaintext
if sub-list 1 and 2 are empty, this is the most often case:
  for each element in sub-list 0:
    if the element is saint:
      if it is a word only:
        if givenname is usually first, store the element as givenname.
        if surname is usually first, store the element as surname.
      else:
        if givenname is usually first, store the element as givenname,
            and store the element after hypen as alternative givenname.
        if surname is usually first, store the element as surname, and
            store the element after hypen as alternative surname.
```
if the element is rubbish:
    pass over it.
if the element is baby-of:
    if givenname is usually first, store the element as givenname.
    if surname is usually first, store the element as surname.
if the element is prefix:
    if givenname is usually first, store the element as givenname.
    if surname is usually first, store the element as surname.
if the element is a number or alphanumeric:
    if givenname is usually first, store the element as givenname.
    if surname is usually first, store the element as surname.
if the element is hyphen:
    if givenname is usually first, store the element as givenname.
    if surname is usually first, store the element as surname.
if the element is a single character:
    it is alternative given name, convert the name usually appear into surname (assuming surname follows after an initial).
if the element is something else:
    all name after hyphen is assigned to the same component.
    if the current word (or hyphened words) is the last and no givenname has been assigned yet, then assign it to givenname.
    if the current word (or hyphened words) is the last and no surname has been assigned yet, then assign it to surname.
    if the current name component is givenname and no givenname has been assigned yet, then assign it to givenname.
    if the current name component is surname and no surname has been assigned yet, then assign it to surname.
    if it no hyphen name and the tag of the current word is givenname (only) then assign to givenname.
    if it no hyphen name and the tag of the current word is surname (only) then assign to surname.
    if the current name component is givenname and the current word is (not only) a surname, then assign current name to givenname.
    if the current name component is surname and the current word is (not only) a givenname, then assign current name to surname.
else:
    if givenname is usually first, store the element as givenname.
    if surname is usually first, store the element as surname.

C.3 Output

The final output from step two are the four names component lists.
The following is the final output using the example input from above.

Given names:        ['del']
Alternative given names: ['van', 'peter']
Surnames:           ['miller']
Alternative surnames: []
The following files are provided with the current distribution of **Febrl** (Version 0.2.2).

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

```plaintext
ANUOS_v1.1.txt
LICENSE.txt
README.txt
address.py
classification.py
classificationTest.py
comparison.py
comparisonTest.py
dataset.py
datasetTest.py
date.py
dateTest.py
encode.py
encodeTest.py
febrl.py
indexing.py
lap.py
lapTest.py
lookup.py
lookupTest.py
mymath.py
mymathTest.py
name.py
output.py
parallel.py
project-deduplicate.py
project-linkage.py
project-standardise.py
randomselect.py
simplehmm.py
simplehmmTest.py
standardisation.py
stringcmp.py
stringcmpTest.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 in the `data/` directory will need to be 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.

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

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

- `address_corr.lst`
- `address_misc.tbl`
- `address_qual.tbl`
- `country.tbl`
- `givenname_f.tbl`
- `givenname_f_freq.csv`
- `givenname_m.tbl`
- `givenname_m_freq.csv`
- `institution_type.tbl`
- `name_corr.lst`
- `name_misc.tbl`
- `name_prefix.tbl`
- `post_address.tbl`
- `postcode_centroids.csv`
- `saints.tbl`
- `surname.tbl`
- `territory.tbl`
- `title.tbl`
- `unit_type.tbl`
- `wayfare_type.tbl`
The `dbgen/` directory contains the database generator `generate.py` and all its associated files. Frequency files are stored in a sub-directory `dbgen/data/`.

- README.txt
- dataset1.csv
- dataset2.csv
- dataset3.csv
- dataset4a.csv
- dataset4b.csv
- generate.py
- data/address1.csv
- data/address2.csv
- data/givenname.csv
- data/postcode.csv
- data/state.csv
- data/streetnumber.csv
- data/suburb.csv
- data/surname.csv
To-Do: Outstanding Development Tasks, Possible Additions and Enhancements

This is an incomplete list of outstanding development tasks and possible additions and enhancements to the Febrl system. Three lists are given, one with issues concerning data cleaning and standardisation, the second with issues on the record linkage or deduplication process, and the third list with Febrl system issues. Note that these lists are not sorted in any particular way.

E.1 Data Cleaning and Standardisation

- Fix problem in HMM address standardisation with warning ‘This should never happen!’ in which case a record is not standardised properly.
- 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.
- 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 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.
- 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/projects/hmm/).
• Explore the use of the forward-backwards algorithm for developing HMMs without explicitly specifying the hidden states.

• 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.

• Provide an improved, more robust version of the tcsv.py delimited file parsing module.

• 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:
Another idea is to use a dictionary of wayfare names in each locality, derived from the phone book or similar 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.

- Exploration of the use of Maximum Entropy Markov Models as described in http://citeseer.nj.nec.com/mccallum00maximum.html instead of HMMs for name and address segmentation. We see this as a very low priority, because HMMs work well enough, but we don’t want to lose the idea of investigating them one day, or someone else investigating them for Febrl, since they are an elegant generalisation of HMMs but very similar. In fact, the only difference is that the emission matrix probabilities are tied to state transitions (edges), not to states (nodes) – hence they depend on the current state and the previous state, not just the current state. This effectively allows states to overlap wrt the observations they emit. Nice!

E.2 Record Linkage and Deduplication

- Implement methods for one2many and many2one assignment restrictions.
- Implement more classifiers, for example the ORLS (Oxford Record linkage Study) classifier, as well as improved machine learning based classifiers.
- Implement expectation-maximisation (EM) algorithm for parameter estimation.
- Include methods to calculate frequency counts from the data used in a linkage process, not just external frequency tables.
- Calculate positive predictive values as presented in the paper Probabilistic record linkage and a method to calculate the positive predictive value by Tony Blakely and Clare Salmond, International Journal of Epidemiology, 2002 31: 1246-1252. Available online at http://ije.oupjournals.org/cgi/content/abstract/31/6/1246?etocPro

E.3 Febrl System

- Repackage the code so it can be installed using the standard Python distutils module.
- Add the functionality to compile proper output data sets, similar to how input and temporary data sets are defined.
• For the display record pairs output form, add the possibility to display the field comparison weights on the right side for the fields that were used in the comparison functions. Currently only the final weight is displayed for each record pair.

• Continue to 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, some unit test will require example data sets to process so that the results can be compared against the expected results.

• 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.

• Add the ability to read and write data from and to ODBC 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.

• Improve parallelism for all modules, make the Febrl system scalable. Explore other parallel environments than PyPar, e.g. PyRO.

• Use Pyrex (or something similar) to speed up execution, see http://www.cosc.canterbury.ac.nz/ greg/python/Pyrex/. An alternative would be to use Psyco, see http://psyco.sourceforge.net/.

• Notes on object orientation of Febrl

1. Add methods so that projects know how to print their own configurations in a neat format, i.e. they write their own documentation.

2. .Save() and .Load() methods should be added to the project class, and these serialise and deserialise 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.

• 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.
Version History

0.1 First public release (6 September 2002)
   Modules for data cleaning and standardisation, hidden Markov model (HMM) training.

0.2 Second public release (14 April 2003)
   Complete object-oriented re-design, includes modules for probabilistic linkage.

0.2.1 Updated second public release (26 June 2003)
   This version is published under an updated ANUOS license version 1.1. Added top level routine standardise in module febrl, added the flexible classifier in module linkage, added a PassFieldStandardiser which allows simple passing of fields without standardisation, added the database generator generate.py, added improved one-to-one assignment based on Auction [2] algorithm, improved memory management and verbose output, added process indicators, added Berkeley database library support in Shelve data set, plus many smaller improvements and bug fixes.

0.2.2 Bug-fix release (November 2003)
   This is a bug-fix release for Febrl 0.2.1. Thanks to everybody who sent us bug reports. The manual is now processed using the Python 2.3.2 documentation system.
Support Arrangements

As stated in the License (see Appendix H), **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 (see title page of this manual).

Two mailing lists have been set up at *Sourceforge.Net*¹. For more details see

https://sourceforge.net/mail/?group_id=62161

If you are interested in this project please subscribe and we will inform you of software releases, publications and other output of this project.

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**.

¹ http://sourceforge.net
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.
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 Modifications(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, MERCHANTABILITY, FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT. THE ENTIRE RISK AS TO
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 declaratory 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.
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 were specified in the identification of the Initial Developer.

APPENDIX 2
DIFFERENCES BETWEEN THE ANUOS LICENSE VERSION 1.1 AND THE ANUOS LICENSE VERSION 1.0

The ANUOS License Version 1.1 was derived from the ANUOS License Version 1.0. The differences between the ANUOS License Version 1.1 (this document) and the ANUOS License Version 1.0 are as follows:

i. In Exhibit A the names of the Australian National University staff members who developed the software were updated to reflect changes in the development team. Specifically, the names of Drs Markus Hegland, Stephen Roberts and Ole Nielsen (Mathematical Sciences Institute, Australian National University) were removed.

ii. In Exhibit A the copyright notice was update from "2002" to "2002, 2003".

135


INDEX

A
address.py, 27, 33
agreement weight, 60
ANU Data Mining Group, 4
ANU Open Source License, 4, 5, 131
approximate string comparator, 9, 10
assignment, 73, 76
  Auction algorithm, 76
  one-to-one, 73, 76
AutoMatch, 41, 97
AutoStan, 41
awk, 41

B
Berkeley database, 92
bigram, 58, 62
bigram index, 58
biomedical research, 3
blocking index, 55, 57
blocking technique, 9
blocking variable, 9
bootstrapping, 43, 48
bsddb3, 92

C
categorical attributes, 27
Centre for Epidemiology and Research, 4
classifier, 55, 68
  Fellegi and Sunter, 68
  flexible, 69
clerical review, 3, 4, 9, 55
clusters of workstations, 4
COL data set, 88
comma separated value, 85, 90, 101, 102
correction.py, 59
correction list, 28, 31, 34, 97, 98
csv data set, 90

data cleaning, 10, 11, 27, 28, 30, 32, 33, 41, 47, 97
data integrity, 9
data items, 27
data matching, 9
data mining, 3, 9, 10
data quality, 9
data scrubbing, 10
data segmentation, 30
data standardisation, 10, 11, 27, 28, 30, 32, 33, 39, 41, 47
data tagging, 28
data warehouse, 10
dataset.py, 85
date format string, 35, 36
date parsing, 35
date standardisation, 35
date.py, 27, 35
deduplication, 9, 55, 71
difflib, 62
disagreement weight, 60
Double-Metaphone, 57, 63
download, 107

e
edit distance, 62
encode.py, 62
epidemiological studies, 3
ethics committee, 3
ETL, 10

F
Fellegi and Sunter, 9, 10
classifier, 68
field comparison function, 55, 59, 61, 66, 67
field passing, 37
fixed column fields, 85, 88
frequency table, 60, 61, 97, 101
fuzzy technique, 10

g
generate.py, 77
geocode table, 97, 102
Index

H
health data, 3
health research, 3
hidden Markov model, 32, 33, 41, 47, 53, 107
  smoothing, 48, 53
    absolute discounting, 48
    Laplace, 48
states, 111
training, 28, 107
Viterbi algorithm, 42–44
histogram, 73

I
indexing.py, 55
information extraction, 41
information retrieval, 10
installation, 107

J
Jaro, 61
join operation, 9

L
list washing, 9
look-up table, 10, 29–31, 34, 42
lookup.py, 66, 97

M
m-probability, 59
machine learning, 4, 9, 10, 28
mailing lists, 9
match weight, 9, 59
maximum likelihood estimate, 47
Memory data set, 94
merge/purge processing, 9
Microsoft Windows, 107
missing weight, 59
Mozilla Public License, 4, 135
MPI, 109
MS-DOS, 107
multiprocessor, 4
MySQL, 85, 91

N
name.py, 27, 30
name_corr.lst, 98
name_misc.tbl, 99
New South Wales Department of Health, 4, 5
Newcombe and Kennedy, 9
NYSIIS, 56, 57, 63
O
object identity, 9
open source, 85
output dataset, 39
output field, 27, 30, 31, 34, 36, 41, 43

P
parallel computing, 4
parallelism, 109
personal attributes, 27
Phonex, 57, 63
pivot year, 36
pre-processing, 9, 10
privacy, 3
probabilistic linkage, 9, 55
project.py, 11, 13, 36, 105, 107
Pypar, 109
Python, 4, 27, 107
  dictionary, 85, 94

R
randomselect.py, 77, 107
record comparator, 55, 66–69
record linkage, 9, 71
record pair, 73
record standardisation, 38
regular expressions, 41
results, 73
rule-based, 30, 35, 41

S
scalar attributes, 27
seqmatch, 62
Shelve data set, 92
shelve.py, 85, 92
simphmm.py, 43, 44
simphmmTest.py, 43
Sleepycat Software, 92
SNOBOL, 41
sorting, 10
sorting index, 57
Soundex, 9, 56, 57, 62
speedup, 5
SQL, 9, 10, 85
SQL data set, 91
standardisation.py, 28, 30, 33, 38, 49
stringcmp.py, 61
supercomputer, 4

T
tag, 29, 30, 42, 97, 99
tagdata.py, 47, 49, 53, 107
tagging table, 97, 99
territory.tbl, 99
text indexing, 10
text segmentation, 41
trainhmm.py, 47, 53, 107
training, 47
training data, 28, 43
transitive closure, 76

U
u-probability, 60
unique identifier, 9
Unix, 107

W
web search engine, 10
weight vector, 55, 59
Winkler, 62
word spilling, 28, 30, 32, 34