Official Statistics Research Series, Vol 5, 2009
ISSN 1177-5017; ISBN 978-0-478-31592-9

The Official Statistics System

Generating Synthetic Microdata from Published Marginal Tables and Confidentialised Files

Alan Lee

Department of Statistics, University of Auckland

Abstract

We describe several methods for generating synthetic data sets, using a combination of publically available marginal tables, and microdata samples. Our methods are based on fitting parsimonious statistical models to high-dimensional tables of relative frequencies, and then generating synthetic data from these models. We describe a set of R functions which implement the methods under study, and apply the methods to data from the 2001 Census of Population and Dwellings.

Keywords

Contingency tables, marginal tables, log-linear models, mixture models, maximum likelihood, iterated proportional fitting, Fisher scoring, latent class analysis, Metropolis-Hastings algorithm.

Citation

Lee, A. (2009). Generating synthetic microdata from published marginal tables and confidentialised files, *Official Statistics Research Series, 5.* Available from www.statisphere.govt.nz/osresearch

Published by

Statistics New Zealand
Tatauranga Aotearoa
Wellington, New Zealand

ISSN 1177-5017 (online)
ISBN 978-0-478-31592-9 (online)

Contents

1 INTRODUCTION

1.1 Confidentiality and microdata access

The confidentiality debate between the research community and national statistics offices is an ongoing one. On the one hand, researchers are always pushing for more access to microdata, in order to perform more and more complex analyses involving more and more variables, while on the other, statistical agencies are concerned to work within their national legislation, and fear that confidentiality concerns will damage public goodwill towards data collection, and undermine compliance.

The two sides of this debate are both motivated by legitimate concerns. The research side can bring powerful arguments to bear. Among these is the idea of an informed citizenry as a guarantee of civil liberties, as neatly expressed in the following quotation:

> "Open access to official statistics provides the citizen with more than a picture of society. It offers a window on the work and performance of government itself, showing the scale of government activity in every area of public policy and allowing the impact of public policies and actions to be assessed."
>
> *1993 UK White Paper on Open Government*

There is also the argument that access improves social policy research and thus is of material benefit to society. Thus, restrictions on access my be socially counterproductive. This is the "privacy paradox", as exemplified by the following quote

> "... people will recognise that while they surely have a right to privacy, they may also come to recognise that they have a duty to share information, if the common good is to be furthered."
>
> *Peter Masden, National Science Foundation 2003 Workshop on Confidentiality Research.*

There is also an economic argument: statistical data is expensive to collect. Allowing researchers outside the statistical agencies access to microdata greatly expands the amount of analysis that can be performed on a given data set, thus getting more value from the data collected.

Balanced against this are the legal and operational guarantees of confidentiality extended by the statistical agencies when data is solicited. The right to confidentiality is enshrined in the sixth of the United Nations Principles of Official Statistics, which states

> "Individual data collected by statistical agencies for statistical compilation, whether or not they refer to natural or legal persons, are to be strictly confidential and used exclusively for statistical purposes."

In recent years, statistical agencies have moved away from a policy of risk avoidance (i.e no disclosure of microdata) to one of risk management (disclosure under controlled

conditions). This recognises that access to microdata can be a social good, and is not in conflict with the quote above if confidentiality can be protected, and if the data is used for the purposes of social research, and not for administrative or commercial purposes. In addition, the access must be consistent with the relevant national legislation, and procedures for access must be transparent.

Several strategies for risk management are practiced by national statistical agencies, most of which use a spectrum of techniques. These stand at various points on the continuum between unfettered access on the one hand, and a strict policy of non-release on the other. These strategies include

Statistical tables (data cubes) These are low-to-medium dimensional tables of cross-classified data, often delivered over the web on demand to the general public, or on a fee paying basis. There is very little disclosure risk for low dimensional tables, but as the number of dimensions rises the risk increases. This can be managed by rounding tables, recoding variables and so on.

Anonymised microdata files These are samples of unit record data, usually released on CD, with identifiers removed and confidentiality enhanced by recoding, and removal or modification of unit records identified as disclosure risks. The risk is further controlled by limiting access to the CD to bona fide social researchers. The degree of confidentialising can be varied according to the restrictions put on dissemination. Files subject to more confidentialising and fewer restrictions are often referred to as public use files.

Remote Access and Data Laboratories An alternative to release of microdata samples is to permit researchers to interrogate the unconfidentialised microdata, but retain the data in a secure environment. Researchers submit analysis requests in the form of computer scripts, which are run on the genuine microdata. The resulting output is vetted by the agency before being released to the researcher. Requests may be submitted to the agency electronically, via internet or email (Remote Access) or researchers may be required to visit a site under the control of the agency (a Data Laboratory).

In this report, we concentrate on the second strategy, that of confidentialised microdata files. The most common method of confidentialising a microdata file is by identifying individuals that may appear as uniques in cross-classified tables of low dimension, and modifying the data for these individuals. This does not of course protect confidentiality completely, since individuals may be identified by their unit records. This can be bombarded in two ways: first by recoding variables into coarser categories (so, for example, an individual's income may be recorded as "over $70,000" rather than the actual figure) and secondly by removing small-area information. However, the disclosure risk remains. The more data is modified, the less reliable inferences can be drawn from it, but the less confidentially risk remains. Developing good quantitative measures of risk and of loss of validity would make this trade off more precise. See Jackson and Gray (2008) for some progress in this area.

Rather than perturbing the original microdata, we focus on another approach. If it were possible to build a probability model for the multivariate distribution underlying the data base, one could then generate purely artificial data (synthetic data) which conveys no risk, but mimics in all essential respects the population under study. This data could then be used, along with multiple imputation techniques, to perform whatever analysis is required. A common approach (Graham and Penny 2007, 2008) is to smooth the empirical distribution derived from the data base and use the smoothed distribution to generate repeated samples for multiple imputation. They use a Bayesian approach for their smoothing method.

Another approach is to divide the variables into “sensitive” and “public” groups. Sensitive variables might be income, date of birth and age, whereas public variables might be sex, occupation or nationality. A model is then fitted to real data using the sensitive variables as responses and the public variables as covariates. The models fitted may be conventional regression models or more data driven models such as CART. The model is then used to impute synthetic values for the sensitive variables, retaining the real values of the public variables. This approach is described in Reiter (2005) and Woodcock and Benedetto (2007).

Our approach in contrast is to assume that researchers will be interested in a subset of variables for their analysis, and it may be possible to model the entire joint distribution of the variables in question by conventional statistical models. These can then be used to generate samples of data as required. The models can be fitted using a combination of publicly available data cubes or other data, be it the genuine microdata or a confidentialised subset thereof.

We require that our models have two attributes:

1. It must be possible to fit the models to reasonably high dimensional tables.

2. It must be possible to simulate from the model efficiently.

We can distinguish several uses for such synthetically generated data sets. At the most basic level, they can be used for educational and training purposes. In this connection, it is desirable that they reproduce at least the low-dimensional marginal behaviour of the real populations which they represent. In a similar manner, synthetic data sets are useful in the development of software programs, particularly those that may be used to interrogate real microdata in a data laboratory setting. At a higher level, if the distribution generating the synthetic data is sufficiently close to the real underlying population distribution, statistical analyses run on the synthetic microdata, when coupled with multiple imputation, will yield results reasonably close to those that would result from using the genuine microdata.

In this report, we consider two classes of statistical models, namely log-linear models and mixture models. We discuss the fitting of these models to data, using both published marginal tables of the whole population and also samples of confidentialised microdata. The present report builds on work described in Lee (2007), which concentrated on fitting log-linear models using published low-dimensional marginal tables. We also address the question of how to sample from the fitted distributions. We illus-

trate the basic methods with three simple examples, taken from the 2001 New Zealand Census of Population and Dwellings.

1.2 Log-linear models

In the analysis of log-linear models for contingency tables, two related sampling scenarios are usually considered. In the first, the cell counts (table entries) are assumed to be independent Poisson variates with specified cell means. Alternatively, we can assume that the individuals being classified are allocated independently to the cells according to a set of cell probabilities, so that the cell counts are realisations of a multinomial distribution. The two sampling schemes are related; the distribution of the Poisson cell counts, conditional on the table total, is in fact multinomial, with cell probabilities being proportional to the Poisson means (i.e. the probability for a cell is the cell mean divided by the total of the cell means.) If we write the cell means as a vector μ, a log linear model for the cell means is one of the form

$$\log \mu = X\beta$$

where X is a design matrix with 0/1 elements, and β is a vector of parameters. Various algorithms (which we discuss in more detail in Chapter 2) can be used to estimate the parameters and hence the cell means, yielding a "fitted mean". We can then generate a simulated or synthetic table by generating a Poisson random variate (having mean equal to the fitted mean) for each cell. If we want the synthetic table to have a fixed total, we can convert the fitted cell means into fitted probabilities by dividing by their totals, and simulate a realisation from the resulting multinomial distribution.

Example 1. To illustrate, consider the following simple example, which concerns three variables from the 2001 Census of Population and Dwellings. The variables, which are measured on the employed usually resident population aged 15 and over, are

EmploymentStatus; Employment Status, having values "Paid Employee", "Self Employed without Employees","Employer", "Unpaid Family Worker", "Not Stated";

Sex; having values "Male", "Female";

WorkLabForceStatus; Work and Labour Force Status, having values "Employed Full-Time", "Employed Part-Time".

The data are shown in Table 1. Suppose we did not have access to the complete table, but only the two-dimensional margins shown in Table 2. As we explain in Chapter 2, knowledge of these three tables permits us to fit a log-linear model to the cell means, in which the cell means are not free, but rather are constrained by a set of constraints of the form

$$\lambda_{ijk} - \lambda_{ij1} - \lambda_{i1k} - \lambda_{1jk} + \lambda_{i11} + \lambda_{1j1}\lambda_{11k} + \lambda_{111} = 0$$

where $\lambda_{ijk} = \log \mu_{ijk}$ and μ_{ijk} is the cell mean corresponding to the cell containing the count of individuals with Employment Status having its ith value, sex having its jth value, and Labour Force Status having its kth value. See Section 2 for more details.

Table 1: Employed usually resident adult population from 2001 census, cross-classified by employment status, sex and labour force status.

WorkLabForceStatus: Full-time

	Sex	
EmploymentStatus	Male	Female
Paid Employee	572,244	425,085
Self-Employed and Without Employees	122,439	40,320
Employer	86,493	29,112
Unpaid Family Worker	9,387	9,831
Not Stated	21,177	12,027

WorkLabForceStatus: Part-time

	Sex	
EmploymentStatus	Male	Female
Paid Employee	75,681	223,905
Self-Employed and Without Employees	20,295	30,063
Employer	3,387	10,638
Unpaid Family Worker	6,765	13,311
Not Stated	5,094	10,014

How closely do these fitted means match the cell counts in Table 1? It is feature of this method that the three two-dimensional marginal tables of cell counts match exactly the marginal fitted counts. What about the complete three dimensional table? The table of fitted means is shown in Table 3, along with the cell counts from Table 1. The agreement is not too bad. Also shown are five synthetic tables, generated from the fitted counts using the Poisson method described above.

To illustrate the simplicity of this method, we show the few lines of R code required to produce this table. The data from Table 1 are in an R data frame called `emp_work_sex.df`, with variables `y` (containing the counts), `EmploymentStatus`, `WorkLabForceStatus` and `Sex`. The code is

```
# fit the model
my.glm = glm(y~(Sex+EmploymentStatus+WorkLabForceStatus)^2, family=poisson,
data=emp_work_sex.df)
# calculate the fitted means
mypred = predict(my.glm, type="response")
# make the synthetic data and Table 3
table1 = cbind(emp_work_sex.df$y, mypred,matrix(rpois(100,mypred),20,5))
#label the table
dimnames(table1) = list(1:20, c("Count", "Fitted mean", paste("Set",1:5)))
```

This method is perfectly adequate for small tables, but is wasteful of memory and cannot cope with large tables having many variables. We do not use the standard

Table 2: Marginal tables for the census data.

	Sex	
EmploymentStatus	Male	Female
Paid Employee	647,925	648,990
Self-Employed and Without Employees	142,734	70,383
Employer	89,880	39,750
Unpaid Family Worker	16,152	23,142
Not Stated	26,271	22,041

	WorkLabForceStatus	
EmploymentStatus	Full-time	Part-time
Paid Employee	997,329	299,586
Self-Employed and Without Employees	162,759	50,358
Employer	115,605	14,025
Unpaid Family Worker	19,218	20,076
Not Stated	33,204	15,108

	WorkLabForceStatus	
Sex	Full-time	Part-time
Male	811,740	111,222
Female	516,375	287,931

R model fitting functions in the rest of this report. Instead, we have written a set of functions that can cope with bigger tables.

1.3 Mixture models

Mixture models (which are also known as latent class models in the social science literature) are an attractive alternative to log-linear models when fitting contingency tables. We illustrate with another example from the 2001 Census of Population and Dwellings.

Example 2. Consider the following data set from the 2001 Census of Population and Dwellings. The variables are

Sex; with levels “Male”, “Female”;

TotalHrsWrkd; Hours Worked, with levels “1-9 Hours”, “10-19 Hours”, “20-29 Hours”, “30-39 Hours”, “40-49 Hours”, “50-59 Hours”, “60 Hours or More”, “Not Elsewhere Included”;

TotalIncomeGroup; Total Personal Income, with levels “Loss or Zero Income”, “$1 - $5,000”, “$5,001 - $10,000”, “$10,001 - $15,000”, “$15,001 - $20,000”, “$20,001 -

Table 3: Actual cell counts and fitted cell means, along with five synthetic tables.

Cell	Count	Fitted mean	Set 1	Set 2	Set 3	Set 4	Set 5
1	572244	573227.079	573271	571998	572048	573537	574317
2	122439	121565.422	122162	121790	121451	122082	122153
3	86493	84295.515	84287	83953	84334	84516	84144
4	9387	11101.534	11207	11186	11228	11162	10863
5	21177	21550.450	21531	21601	21468	21468	21693
6	425085	424101.921	423698	424377	424174	424639	424097
7	40320	41193.578	40977	40961	41665	41159	40902
8	29112	31309.485	31439	31589	31221	31324	31523
9	9831	8116.466	8089	8088	8115	8155	8163
10	12027	11653.550	11649	11791	11640	11669	11693
11	75681	74697.921	75134	74771	74455	74570	75322
12	20295	21168.578	21032	21158	21034	21016	21179
13	3387	5584.485	5640	5651	5639	5659	5539
14	6765	5050.466	5016	5072	5066	4962	5052
15	5094	4720.550	4706	4793	4648	4753	4763
16	223905	224888.079	225273	224706	224819	226059	224956
17	30063	29189.422	29407	29093	29180	29032	29122
18	10638	8440.515	8344	8368	8514	8608	8414
19	13311	15025.534	15037	15096	14883	15118	14985
20	10014	10387.450	10218	10489	10441	10386	10523

$25,000", "$25,001 - $30,000", "$30,001 - $40,000", "$40,001 - $50,000", "$50,001 - $70,000", "$70,001 or More", "Not Stated".

In a mixture model, we model the cell probabilities as a mixture of T independent tables. Thus, in our example, with π_{ijk} the cell probability for the ith value of `Sex`, the jth value of `TotalHrsWrkd` and the kth value of `TotalIncomeGroup`, we have

$$\pi_{ijk} = \sum_{t=1}^{T} \tau_t \alpha_{it} \beta_{jt} \gamma_{kt},$$

where for each t, $\{\alpha_{it}\}$, $\{\beta_{jt}\}$ and $\{\gamma_{kt}\}$ are probability distributions, and $\{\tau_t\}$ is the *mixture distribution*. The idea is that the population can be divided into T unobserved classes, and conditional on class membership, the variables are independent.

These models have four desirable features for our purposes. First, they offer a flexible class of models, whose complexity is easily adjusted by the choice of T, the number of components in the mixture. Second, they are easy to fit to quite high dimensional tables, using the EM algorithm, or, equivalently, by solving the score equations by the method of functional iteration. Third, they are easy to simulate from: we select a class t using the mixture distribution $\{\tau_t\}$, and then simulate the variables independently using

the distributions $\{\alpha_{it}\}$, $\{\beta_{jt}\}$ and $\{\gamma_{kt}\}$. Finally, it is easy to calculate any marginal distribution: we simply take the same mixture of independence tables corresponding to the marginal distribution. For example, the marginal distribution of `Sex` and `TotalHrsWrkd` is

$$\pi_{ij+} = \sum_{t=1}^{T} \tau_t \alpha_{it} \beta_{jt}.$$

We can fit the mixture model using SAS PROC LCA, assuming that the data are contained in a SAS data set `example2` of the following form: There is a line in the data set for each of the $12 \times 8 \times 2 = 192$ factor level combinations, and the variables are `count`, containing the cell count, and `tpincome`, `hourswkd` and `sex`, represented as integers. The code to fit a model with five components is

```
proc lca data=example2;
nclass 5;
items sex totalhrswrkd totalincomegroup;
categories 2 8 12;
freq count;
seed 45687435;
run;
```

In practice, we use an R function which can handle bigger problems; see Chapter Three for details. The EM algorithm used has a tendency to become trapped in local maxima, so the algorithm is usually repeated with different starting values to find a more reliable solution. In Table 4, we show the deviance (G^2) for fitting models with various numbers of components to these data, together with the deviance of a log-linear model of the type fitted in Example 1. The deviance is a measure of discrepancy between the table counts and the fitted model: if the fitted probabilities are $\hat{\pi}_{ijk}$ and the counts are n_{ijkj} then the deviance is

$$G^2 = 2 \sum_{ijk} n_{ijk} \log \left(\frac{n_{ijk}}{n \hat{\pi}_{ijk}} \right),$$

where n is the table total. As the table indicates, and as theory predicts, as the model becomes more complex and the number of parameters increases, the goodness of fit as measured by the deviance also increases. Thus, the mixture model allows us to trade off complexity and fit in a flexible way.

Table 4: Number of components, deviances, and numbers of parameters

Number of components	Number of parameters	Deviance
5	99	25,224.02
8	159	3,132.51
10	199	895.99
Log-linear	115	9,642.69

1.4 Fusing data sets using mixture models

Mixture models are also useful for fusing two data sets together. Suppose we have two categorical data sets A and B, which have a set of Q variables in common. In addition, A has P variables that are not included in B, and B has R variables not included in A. If the joint distribution of the $P + Q + R$ variables can be described by a mixture model, then we can fit the model using A and B alone, using a variant of the EM algorithm. This gives us a way of joining or fusing the data sets together. The method was introduced by Kamakura and Wedel (1997) in the case of two datasets, but the same method can be used to fuse multiple data sets provided the degree of overlap is sufficient. This is examined in more detail in Sections 2 and 3.

Example 3. To illustrate the procedure, consider the following two data sets from the 2001 census CURF: the first is the one used in Example 2, using the variables `Sex`, `TotalHrsWrkd` and `TotalIncomeGroup`, while the second consists of the variables `Sex` and `TotalIncomeGroup`, along with the variable `OccupationCode`, which has ten possible values: Elementary Occupations; Legislators, Administrators and Managers; Professionals; Technicians and Associate Professionals; Clerks; Service and Sales Workers; Agriculture and Fisheries Workers; Trades Workers; Plant and Machine Operators and Assemblers; Not Elsewhere Included.

The four-variable model is fitted with the R function `mixture.fitter.fusion`, described in Chapter 3. How well do the margins of the fitted distribution match the data sets A and B? In Figure 1.4, we display two Pareto plots of the biggest 30 cell frequencies and the corresponding fitted probabilities, one for data set A and one for B. The bars are arranged in order of descending relative frequency. The agreement is good.

1.5 Data sources and software

In this report, we use data taken from the web pages of Statistics New Zealand. These data are available via the Tablebuilder facility, which allows members of the public to download low-dimensional tables of data. We use tables derived from the 2001 Census of Population and Dwellings. In addition, we use data from the 2001 census CURF: this is a 2% sample from the 2001 census that has been confidentialised using the methods described above in Section 1.1. The corresponding data from the 2006 census is also available using tablebuilder, but as no 2006 census CURF is currently available, we have restricted ourselves to the 2001 data.

We use the R statistical environment (R Development Core Team, 2008) for our work, running under Windows. Although other packages, such as SAS, can be used to fit both log-linear and mixture models, they cannot cope with the large tables we consider. Some familiarity with R is assumed in the report.

1.6 Organisation of the report

The simple examples discussed above illustrate the methods we will use to construct synthetic data sets, using both published tables and the 2001 census CURF. In the

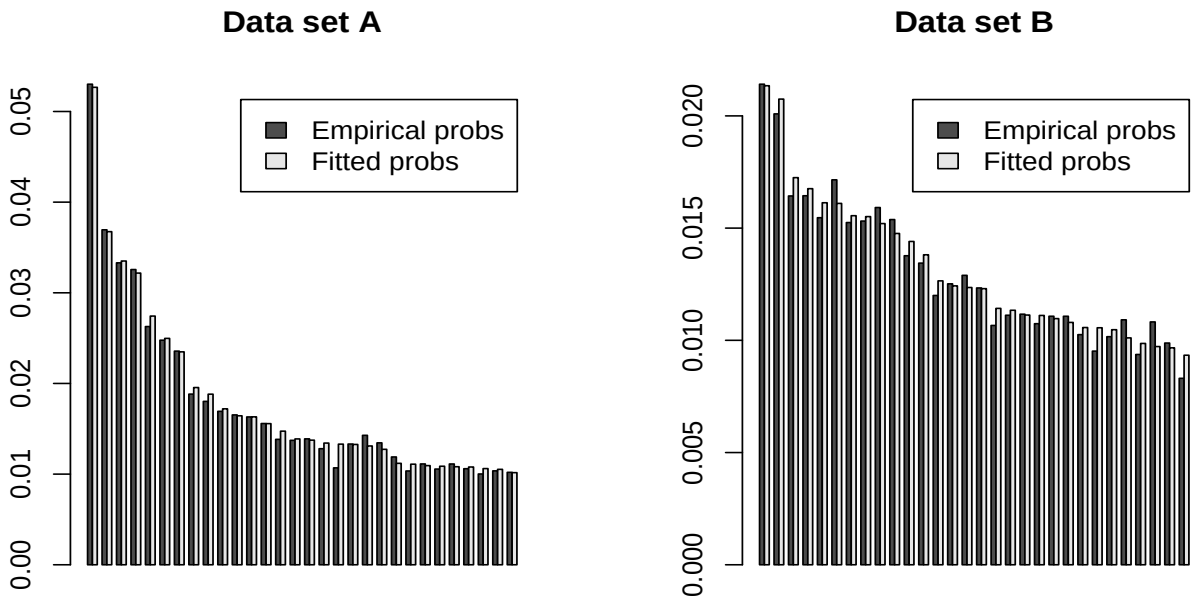

Figure 1: Pareto plots for the data fusion example.

rest of the report, we discuss the software and algorithms we require to solve problems of a more realistic size. In Section 2, we outline the relevant theory, beginning with a discussion of log-linear models. We develop a general notation suitable for dealing with tables of arbitrary size, and discuss the connections between log-linear models, marginal tables and the IPF algorithm. We discuss how a trivial modification of the algorithm can be used to accept data from various sources, and make some remarks about simulating data from these models. We then go on to describe the mixture model, how it can be fitted by solving the score equations, and how it can be modified to fuse several different data sets together. We also discuss generating data from such models. Finally, we make some remarks on fitting both kinds of model to the CURF data alone.

In Section 3, we describe the algorithms and software we use to generate our synthetic data. We begin by describing the data structures we use to represent tables, and then discuss some R functions that can be used to fit log-linear models using the IPF algorithm. Next, we give a description of an R mixture model implementation.

In Section 4, we describe a set of marginal tables from the 2001 Census of Population and Dwellings, and show how they, along with the census CURF, can be used to generate a variety of synthetic data sets. Summary and conclusions are in Section 5, and more formal documentation of the software appears in the Appendices.

2 THEORY

In this section we outline the theory that underpins our software, and develop a notation suitable for discussing contingency tables of arbitrary size.

2.1 Contingency tables

Suppose we have a population of N individuals, on each of whom we make K categorical measurements. Typical examples of such measurements or *factors* are age groups, employment status and so on. We denote these factors by $A_1, \ldots, A_K$. The set of possible categories for a factor is called the set of *levels* for that factor. Thus, the factor "Gender" has levels ("Female", "Male"), and the factor "Age group" has levels (0-4, 5-9, ..., 80-84, 85+), or any other definition that might be appropriate. We assume that there are a finite (typically small) number of levels for each factor. We denote the number of levels of factor A_k by I_k. Thus, there are $I = I_1 \times I_2 \times \cdots I_k$ possible combinations of levels. A typical combination of levels is denoted by $i = (i_1, \ldots, i_K)$.

The usual way to represent the data on these N individuals is by a *contingency table*, an $I_1 \times I_2 \times \cdots I_k$ array of counts, where the array element in position $(i_1, \ldots, i_K)$ or "cell count", is the number out of the N individuals that have $A_1 = i_1, \ldots, A_K = i_K$. We use the notation $y[i_1, \ldots, i_k]$ or more compactly $y[i]$ to denote the cell count.

Note that the table depends on the order of the factors, and the ordering of the levels of each factor. Given a fixed order of the factors, (which in this report will usually be alphabetical), and a fixed ordering of the levels of each factor, we can arrange the cell counts in a one-dimensional array or *vector* by stringing the counts out in reverse lexicographic order, where the leftmost index i_1 varies most rapidly, followed by i_2 and so on. Thus if $K = 2$, $I_1 = 2$, and $I_2 = 3$, the ordering would be

$$y[1,1],\ y[2,1],\ y[1,2],\ y[2,2],\ y[1,3],\ y[2,3].$$

In some contexts the array representation is more convenient, and in others the vector representation.

2.1.1 Marginal tables

Given a contingency table, we can form various marginal tables by summing over certain indices. For example, suppose we have a 3-dimensional table with three factors, say age group, employment status and gender. We can form the marginal age group $\times$ employment status table by summing over the index corresponding to gender. The marginal table has counts

$$y_M[i_1, i_2] = \sum_{i_3 = 1, \ldots, I_3} y[i_1, i_2, i_3].$$

In a similar manner, we can form the gender $\times$ employment status table by summing over age group, and the age group $\times$ employment status table by summing over gender. We also have one-dimensional tables: the gender table is formed by summing

over age group and employment status, and so on. We can also regard the table grand total as a"null margin" formed by summing over all the factors. It is clear that to every subset S of $\{1, \ldots, K\}$ there corresponds a marginal table formed by summing over all the indices *not* in S. Thus, there are 2^K possible marginal tables, including the original table and the table total.

2.2 Log-linear models

In contingency table work, a common assumption is to regard every count in the table as the realization of a Poisson random variable with a given mean μ. A *model* for the table is a formula which specifies the mean μ as a function of the factor levels corresponding to each cell. We write μ as $\mu[i_1, \ldots, i_K]$ to emphasise the dependence on the factor levels. Since the Poisson means are necessarily positive this constraint is neatly handled by specifying the log of the mean. Such models, giving the form of $\log \mu[i_1, \ldots, i_K]$, are called log-linear models. Excellent discussions of these models are to be found in Agresti (2002) and Christensen (1997).

We now describe a useful class of log-linear models that correspond in a natural way to classes of marginal tables. We begin by considering the case where the table is two-dimensional, so that we have $K = 2$ and two factors A_1 and A_2. Put $\lambda[i_1, i_2] = \log \mu[i_1, i_2]$, and define four sets of parameters as follows.

The "constant term" : this is $\beta_0 = \lambda[1, 1]$.

The "A_1 main effects" : There are I_1 of these, defined by $\beta_1[i_1] = \lambda[i_1, 1] - \lambda[1, 1]$, $i_1 = 1, 2, \ldots, I_1$. Note that necessarily $\beta_1[i_1] = 0$ by definition.

The "A_2 main effects" : There are I_2 of these, defined by $\beta_2[i_2] = \lambda[1, i_2] - \lambda[1, 1]$, $i_2 = 1, 2, \ldots, I_2$. Again, $\beta_2[1] = 0$ by definition.

The "A_1A_2 interactions" : There are $I_1 \times I_2$ of these, defined by $\beta_{12}[i_1, i_2] = \lambda[i_1, i_2] - \lambda[i_1, 1] - \lambda[1, i_2] - \lambda[1, 1]$, $i_1, i_2 = 1, 2, \ldots, I_2$. Note that $\beta_{12}[i_1, i_2] = 0$ if either of i_1 or i_2 is 1.

In terms of these quantities, we can write

$$\lambda[i_1, i_2] = \beta_0 + \beta_1[i_1] + \beta_2[i_2] + \beta_{12}[i_1, i_2].$$

Note that this parametrization puts no restrictions on the means $\mu[i_1, i_2]$: we have simply expressed the $I_1 \times I_2$ means in terms of $I_1 \times I_2$ non-zero new parameters (1 constant term, $(I_1 - 1)$ non-zero A_1 main effects, $(I_2 - 1)$ A_2 main effects, and $(I_1 - 1) \times (I_2 - 1)$ non-zero A_1A_2 interactions. By arranging the constant terms, main effects and interactions into a vector β, we can write

$$\log \mu = X\beta, \tag{1}$$

where X is the *model matrix*.

By setting the elements of β corresponding to the interactions to zero, we obtain a new, restricted model for the cell means.

2.2.1 The general case

Suppose now we have a K-dimensional table, obtained by classifying the individuals in the population according to K criteria $A_1, \ldots, A_K$. For $l = 1, \ldots, K$ and $j = 2, .., I_l$, define a "dummy variable" $D_j^{(l)}$ by

$$D_j^{(l)}[i_1, \ldots, i_K] = \begin{cases} 1, & \text{if } i_l = j; \\ 0, & \text{otherwise.} \end{cases}$$

For a subset $\{l_1, \ldots, l_r\}$ of $\{1, \ldots, K\}$, let $X_{l_1,\ldots,l_r}$ be the matrix whose $I_1 \times I_2 \times \cdots \times I_K$ rows correspond to the different factor level combinations $(i_1, \ldots, i_K)$, and whose columns are of the form

$$D_{j_1}^{(l_1)} \times D_{j_2}^{(l_2)} \cdots \times D_{j_r}^{(l_r)}, \qquad 2 \le j_1 \le I_{l_1}, \ldots, 2 \le j_r \le I_{l_r},$$

where $\times$ denotes element-wise multiplication. Then put

$$X = \big[1|X_1|\cdots|X_K|X_{12}|\cdots|X_{(K-1),K}|\cdots|X_{1,\ldots,K}\big].$$

Thus, for example when $K = 3$,

$$X = \big[1|X_1|X_2|X_3|X_{12}|X_{13}|X_{23}|X_{123}\big],$$

where 1 is a column of ones. We can show that the matrix X is square, non-singular and has $I_1 \times I_2 \times \cdots \times I_K$ rows and columns, so that there is a vector β such that

$$\log \mu = X\beta. \tag{2}$$

We call X the *saturated model matrix*.

If we partition the vector β conformably with X, we obtain subvectors $\beta_{l_1,\ldots,l_r}$ corresponding to the submatrices $X_{l_1,\ldots,l_r}$. We call the elements of $\beta_{l_1,\ldots,l_r}$ the $A_{l_1}A_{l_1}\cdots A_{l_r}$ interactions. By setting various of the subvectors $\beta_{l_1,\ldots,l_r}$ equal to zero, we get various constrained sets of cell means. We can think of a particular model $\mathcal{M}$ as being specified by a particular set of non-zero interactions. The model matrix for the model $\mathcal{M}$ is the matrix $X_{\mathcal{M}}$ obtained by deleting the blocks corresponding to the zero interactions from the saturated model matrix X. Thus, the cell means specified by our model $\mathcal{M}$ are given by

$$\log(\mu) = X_{\mathcal{M}}\beta_{\mathcal{M}}, \tag{3}$$

where $\beta_{\mathcal{M}}$ is β with the zero interactions deleted.

Example 4. Consider the case $K = 2$, $I_1 = 2$, $I_2 = 3$. Then the dummy variables are

cell	$D_2^{(1)}$	$D_2^{(2)}$	$D_3^{(2)}$
11	0	0	0
21	1	0	0
12	0	1	0
22	1	1	0
13	0	0	1
23	1	0	1

and the matrix X is

$$\left|\begin{array}{c|c|c|c|c|c} 1 & D_2^{(1)} & D_2^{(2)} & D_3^{(2)} & D_2^{(1)}D_2^{(2)} & D_2^{(1)}D_3^{(2)} \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \end{array}\right|,$$

where the blocks in the matrix correspond to the partition $\beta_0,\ \beta_1,\ \beta_2,\ \beta_{12}$ of β into constant term, A_1 main effects, A_2 main effects, and A_1A_2 interactions.

We will assume that the log-linear models we consider are *hierarchical*, in the sense that if $\beta_{l_1,\ldots,l_r}$ is non-zero, so is the β corresponding to any subset of $\{l_1,\ldots,l_r\}$. Thus, if in a hierarchical model β_{12} is non-zero, then β_1 and β_2 must be non-zero as well. A model for which all the $\beta_{l_1,\ldots,l_r}$'s are non-zero is called the *saturated* model, and the model for which all the $\beta_{l_1,\ldots,l_r}$'s are zero (except for the constant term) is called the *null* model. The saturated model puts no restrictions on the cell means, but all other models do.

Since we are assuming that our models are hierarchical, we do not need to specify all the non-zero interactions, but only the maximal ones. Thus, if a hierarchical model has an $A_1A_2A_3$ interaction, we don't need to explicitly list the A_1A_2, A_1A_3 and A_2A_3 interactions, they are included implicitly. Thus, we can specify our models compactly by listing only these maximal interactions. In this report, we adopt the notation used by the R statistical system where, for example, the saturated model for 3 factors, with maximal interaction $A_1A_2A_3$, is written $A_1 * A_2 * A_3$, and the model with the $A_1A_2A_3$ interactions zero, with maximal interactions A_1A_2, A_1A_3 and A_2A_3, is written $A_1 * A_2 + A_1 * A_3 + A_2 * A_3$. An alternative notation often used in textbooks, is the "square bracket" notation. Using this, the saturated model is written $[A_1A_2A_3]$ and the model with the zero 3-factor interaction is written $[A_1A_2][A_1A_3][A_2A_3]$.

2.2.2 Estimation

To estimate the parameters of a model from a complete table, we use the method of maximum likelihood. Let $i = (i_1,\ldots,i_K)$ denote a typical cell. The cell count is $y[i]$ and the cell mean is $\mu[i] = \exp(\lambda[i])$, where $\lambda[i] = X_{\mathcal{M}}[i]^T\beta_{\mathcal{M}}$, and $X_{\mathcal{M}}[i]^T$ is the row of $X_{\mathcal{M}}$ corresponding to cell i. The log-likelihood is

$$\begin{aligned} l &= \sum_i y[i]\log\mu[i] - \mu[i] \\ &= \sum_i y[i]\lambda[i] - \exp(\lambda[i]). \end{aligned}$$

To estimate the parameters, we must maximize l as a function of $\beta_{\mathcal{M}}$. Consider a typical element a of $\beta_{\mathcal{M}}$, corresponding to a column $D_{j_1}^{l_1}\cdots D_{j_r}^{l_r}$ of $X_{\mathcal{M}}$. Then, since

$\lambda[i] = X_{\mathcal{M}}[i]^T \beta_{\mathcal{M}}$, we get

$$\frac{\partial \lambda[i]}{\partial a} = D_{j_1}^{l_1}[i] \cdots D_{j_r}^{l_r}[i]$$

so that

$$\begin{aligned} \frac{\partial l}{\partial a} &= \sum_i \frac{\partial l}{\partial \lambda[i]} \frac{\partial \lambda[i]}{\partial a} \\ &= \sum_i (y[i] - \exp(\lambda[i])) D_{j_1}^{l_1}[i] \cdots D_{j_r}^{l_r}[i] \end{aligned}$$

so that at the maximum we get

$$\sum_i y[i] D_{j_1}^{l_1}[i] \cdots D_{j_r}^{l_r}[i] = \sum_i \mu[i] D_{j_1}^{l_1}[i] \cdots D_{j_r}^{l_r}[i]. \quad (4)$$

The expression on the left is just

$$\sum_{i_{l_1}=j_1,\ldots,i_{l_r}=j_r} y[i],$$

which is the $j_1, \ldots, j_r$ entry in the marginal table for $A_{l_1}, \ldots, A_{l_r}$. It follows that to estimate the parameters, we don't require the original complete table of counts, but only the marginal tables corresponding to the non-zero interactions in the model. Thus, for example, to fit the model $A_1 * A_2 + A_1 * A_3 + A_2 * A_3$, we require only the A_1A_2, A_1A_3 and A_2A_3 marginal tables. We can then obtain the fitted mean counts for each cell of the complete table by solving the equations (4).

The standard statistical algorithm for solving the equations (4) is known as iterated proportional fitting or IPF, and was invented in the 1940's by Deming and Stephan. We describe the algorithm briefly in the next subsection.

2.2.3 The IPF algorithm

The IPF is a simple but effective algorithm, which allows us to compute the fitted cell means corresponding to a given set of marginal tables, without having to compute the maximum likelihood estimates of β. It was introduced by Deming and Stephan (1940) and has been adapted to many other uses besides the fitting of log-linear models to contingency tables. The algorithm is as follows:

Step 1: Set $\mu[i] = 1$ for each cell i.

Step 2: For each margin in turn, update the $\mu[i]$'s by adjusting them so that the marginal table of fitted means matches the marginal table of cell counts. i.e. for the $l_1, \ldots, l_r$ margin, update the $\mu[i]$'s using the equation

$$\mu^{NEW}[i] = \mu^{OLD}[i] \times \frac{y_{l_1,\ldots,l_r}[i_{l_1}, \ldots, i_{l_r}]}{\mu^{OLD}_{l_1,\ldots,l_r}[i_{l_1}, \ldots, i_{l_r}]}.$$

Here, $y_{l_1,\ldots,l_r}[i_{l_1}, \ldots, i_{l_r}]$ is the $i_{l_1}, \ldots, i_{l_r}$ entry in the $l_1, \ldots, l_r$-margin of counts, and $\mu^{OLD}_{l_1,\ldots,l_r}[i_{l_1}, \ldots, i_{l_r}]$ is the corresponding entry in the $l_1, \ldots, l_r$-margin of fitted mean counts.

Step 3: Repeat Step 2 until the process converges.

Example 5. For $K = 3$, to fit the model $A_1 * A_2 + A_1 * A_3 + A_2 * A_3$, step 2 takes the form:

Step 2a: Adjust the means to match the A_1A_2 margin by

$$\mu^{NEW}[i_1, i_2, i_3] = \mu^{OLD}[i_1, i_2, i_3] \times \frac{y_{1,2}[i_1, i_2]}{\mu_{1,2}^{OLD}[i_1, i_2]},$$

where $y_{1,2}[i_1, i_2] = \sum_{i_3} y[i_1, i_2, i_3]$.

Step 2b: Adjust the means to match the A_1A_3 margin by

$$\mu^{NEW}[i_1, i_2, i_3] = \mu^{OLD}[i_1, i_2, i_3] \times \frac{y_{1,3}[i_1, i_3]}{\mu_{1,3}^{OLD}[i_1, i_3]}.$$

Step 2c: Adjust the means to match the A_2A_3 margin by

$$\mu^{NEW}[i_1, i_2, i_3] = \mu^{OLD}[i_1, i_2, i_3] \times \frac{y_{2,3}[i_2, i_3]}{\mu_{2,3}^{OLD}[i_2, i_3]}.$$

The algorithm converges reasonably quickly. The log-likelihood is increased at each step. An R implementation of the algorithm is described in Section 3.2.1. To implement the algorithm efficiently in R requires that the table be stored in the memory of the computer. In addition, working storage equal to the table is required to achieve adequate speed. Details of the R implementation of the algorithm are given in Section 3.2.1.

2.2.4 A modified algorithm

Practically speaking, to fit the cell means we simply take each supplied margin in turn, and adjust the current fitted table so that the margin of the fitted table matches the supplied margin. This will work even if the supplied margins are not all margins calculated from the original microdata. For example, suppose we want to fit a model corresponding to fitting all three-factor interactions, but only a subset of these are available for the full population. We can supplement these with the remaining three-dimensional marginal tables calculated from the CURF, or what other source of confidentialised microdata is available. While there is no guarantee that the IPF method will converge in this case, the method works well in practice.

2.2.5 Fitting log-linear models using Fisher scoring

The Fisher scoring method for fitting log-linear models may be a practical alternative to IPF in situations where the number of the cells is large but the number of parameters is modest in comparison, say a few thousand. The limiting factor in fitting models using IPF is the need to manipulate the complete table, which can have a very large number

of cells. As noted above, the algorithm we have implemented requires the storage of two complete tables. If this requirement cannot be met, a method that does not require the storage of a complete table may be preferred. Such a method is Fisher scoring (equivalent in this case to the Newton-Raphson algorithm), which unlike the IPF method, we can easily implement in R without requiring the whole table be stored in the R workspace. Suppose the log-linear model is

$$\log \mu = X\beta.$$

If we assume a Poisson model, as noted in Section 2.2.2, the log-likelihood is

$$l = \sum_i \{y[i]x[i]^T\beta - \exp(x[i]^T\beta)\}$$

where we have written the row of X corresponding to cell i as $x[i]$. Note that $x[i]$ can be calculated in terms of the quantities $D_j^l[i]$ as discussed in Section 2.2.1. The score vector and information matrix are

$$s(\beta) = \sum_i x[i]^T\{y[i] - \exp(x[i]^T\beta)\}$$

and

$$I(\beta) = \sum_i x[i]x[i]^T \exp(x[i]^T\beta).$$

The maximum likelihood estimates are the solutions of the equation $s(\beta) = 0$, and can be calculated using the iterations

$$\beta_{new} = \beta_{old} + I^{-1}(\beta_{old})s(\beta_{old}).$$

This involves forming $s(\beta)$ and $I(\beta)$ at each stage, solving the linear equations $I(\beta)\delta = s(\beta)$ for δ and updating β_{new} by $\beta_{old} + \delta$. The solution of the linear equations is routine for a parameter vector with a few thousand elements. The formation of the score and information matrix can be done by reading in the counts from a external file in blocks, and calculating the rows $x[i]$ sequentially using the representation in terms of the $D_j^l[i]$'s. This is facilitated by arranging the counts in standard form (i.e. in reverse lex order). The calculation is particularly simple in the case of the two or three-factor interaction model. More details of the computer algorithm are given in Chapter 3.

2.2.6 Generating data from log-linear models

Suppose we have fitted a log-linear model and have calculated the table of fitted means, and the corresponding table of probabilities $\{\pi[i]\}$(i.e. by normalising the mean counts so they add to 1.) . How can we generate a synthetic data set using these?

If we do not want to restrict the total sample size, we can simply draw a random count for each cell from a Poisson distribution. If we want a synthetic data set of size approximately N, we draw the cell count for cell i from a Poisson distribution with mean $N\pi[i]$. If, on the other hand, we want a synthetic data set of size exactly N,

we repeatedly select a cell at random with replacement from the set of cells, so that cell i is chosen with probability π_i. This process is repeated until the desired data set size is achieved. Equivalently, we can make a single selection from the multinomial distribution with parameters N and $\{\pi\}$.

To select a cell at random according to the probabilities $\{\pi[i]\}$, we can use the inversion method (see e.g. Ripley 1987). To generate a cell i according to the $\{\pi[i]\}$, we first order the cells (how is immaterial, but reverse lex order is convenient). Then generate a random $U[0,1]$ deviate U, and select cell i, where

$$\sum_{j<i} \pi[j] < U \leq \sum_{j \leq i} \pi[j].$$

Several R functions exist to perform these tasks, and their use is illustrated in Chapter 3. The function `rpois` generates Poisson variates, and the function `sample` implements the rejection method. The function `rmultinom` generates deviates with a multinomial distribution. Note that all these approaches in R assume that the complete table can be stored in memory.

2.2.7 Generating data from log-linear models using the Metropolis–Hastings algorithm

In this section, we outline a method for simulating data from a log-linear model without calculating the fitted cell means, but rather using the estimates of β from the Newton-Raphson algorithm. Suppose we have a stochastic mechanism (technically a Markov Chain) that skips from one cell of the table to another, governed by a *transition matrix* of the form $P = (p_{ij})$ where

$$p_{ij} = Pr(\text{Skip to cell } j | \text{Start in cell } i).$$

Thus, p_{ij} is the probability we will skip to cell j, given we are currently in cell i. Suppose we start this process running. Ultimately, under certain conditions on the p's, the distribution of the cells will stabilize to a distribution called the *stationary distribution* of the chain. This is

$$\pi[i] = \lim_{t \to \infty} Pr(\text{In cell } i \text{ after } t \text{ skips}).$$

Thus, to sample from the distribution π, we let the chain run and after a "burn in" period, the sequence of skips should look like a sample from the stationary distribution. The sample values are not independent, but if we "thin the chain" and retain only every 10th (or 100th, or 1000th) observation, the thinned chain will look very like a random sample from π.

Turning this around, can we construct a Markov chain so that the stationary distribution is some specified distribution π? This amounts to defining a suitable transition matrix P. The Metropolis–Hastings algorithm (Hastings 1970) allows us to do this. Suppose that Q is a transition matrix of some arbitrary chain consisting of skips in our table. This means that Q is an $I \times I$ matrix with positive entries whose row sums are

1. (Recall that I is the number of cells in our table.) We will also assume that $\pi[i] > 0$ for each cell i. For each pair if cells i, j define

$$a_{ij} = \min\left(1, \frac{\pi[j]Q_{ji}}{\pi[i]Q_{ij}}\right).$$

Then the desired transition matrix P has elements $p_{ij} = \alpha_{ij}Q_{ij}$. To sample a new value from the resulting chain, we can use the following:

Step 1: Suppose we are in cell i. Select a cell j at random using the probabilities Q_{ij}.

Step 2: Compute $\alpha_{ij} = \min(1, \pi[j]Q_{ji}/\pi[i]Q_{ij})$.

Step 3: Draw a uniform random number U. If $U < \alpha_{ij}$, skip to cell j, otherwise remain at cell i.

This sequence is repeated until the desired number of values have been generated.

To implement this method, we need to come up with a suitable Q, so that sampling from the distribution $\{q_{ij}\}$ is easy for each fixed i, and does not require that we store Q in the computer's memory. (Storing Q is impossible for all but small tables as the number of entries in Q is $I \times I$.) A simple choice of Q is to assume that the rows of Q are identical, and correspond to the table where the variables are all independent, with one-dimensional marginal distributions matching the marginal distributions of π. Thus,

$$q_{ij} = q_j = Pr(A_1 = j_1) \times Pr(A_2 = j_2) \times \cdots \times Pr(A_K = j_k).$$

The sampling process works best when the rows of Q are as close as possible to π.

We are assuming that we have not computed the $\pi[i]$'s explicitly. How then can we calculate α_{ij}? Note that α_{ij} involves only the ratio $\pi[j]/\pi[i]$. This is much easier to compute than $\pi[j]$ since to compute $\pi[j]$ we must sum the fitted cell means to get the normalising constant. Suppose we have fitted a log-linear model with all two-factor interactions, so that we have estimates of the constant term, the main effects and the two-way interactions. Call these β_0, $\beta^{(k)}_{i_k}$ and $\beta^{(k,l)}_{i_k,i_l}$ respectively, so that

$$\log \mu[i] = \beta_0 + \sum_{k=1}^{K} \beta^{(k)}_{i_k} + \sum_{1 \le k < l \le K} \beta^{(k,l)}_{i_k,i_l}.$$

Then

$$\begin{aligned}
\log(\pi[j]/\pi[i]) &= \log(\mu[j]/\mu[i]) \\
&= \sum_{k=1}^{K} \beta^{(k)}_{j_k} + \sum_{1 \le k < l \le K} \beta^{(k,l)}_{j_k,j_l} \\
&\quad - \sum_{k=1}^{K} \beta^{(k)}_{i_k} - \sum_{1 \le k < l \le K} \beta^{(k,l)}_{i_k,i_l}
\end{aligned}$$

Thus, to compute α_{ij}, we need only know the estimates $\beta^{(k)}_{j_k}$ and $\beta^{(k,l)}_{j_k,j_l}$ and the one-dimensional marginal probabilities.

2.2.8 Convergence and confidentiality issues

The convergence of the IPF method is guaranteed provided all the supplied marginal tables have positive entries. It is obvious that any zero cell in any margin will introduce a zero cell in the fitted table, and that once introduced, this zero cell will persist. In this case, convergence to the MLE cannot be guaranteed. However, this seems not to be a problem in practice, as we are merely trying to find a fitted table close to the true table.

However, there is a confidentiality issue here. Suppose we use the CURF alone to fit a reasonably high-dimensional model using IPF. Since the 2001 Census CURF is a 2% sample, many of the marginal distributions will contain sampling zeros. The effect is to zero out many fitted cell probabilities. In an extreme case, this has the effect of confining the support of the fitted distribution (i.e. the cells with non-zero-fitted probabilities) to the factor-level combinations of individuals actually present in the CURF. Thus, sampling from the fitted distribution will be very similar from merely resampling the lines of the CURF, and represent no advance in confidentiality over that in the CURF itself. Even if we use a bigger sample of microdata, the problem will persist if we try to fit bigger and bigger models.

The case is different with structural zeroes. Any structural zeroes in the marginal distribution will be preserved in the fitted probabilities, which is a desirable feature of the method.

One way to deal with the problem of sampling zeroes is to smooth the CURF before fitting a model. Suppose we take a mixture of the CURF and an independence model as follows: Given a set of variables, form the corresponding contingency table from the CURF. This will likely contain many sampling zeros. For cell i of the table, let $y[i]$ be the cell count. Then fit an independence model to the table. As long as none of the one-dimensional tables has any sampling zeroes, the independence model will have no zero estimated cell counts. The fitted cell probability corresponding to the independence model is

$$\pi^{(IND)}[i] = \frac{y[i_1 + \cdots +]}{n} \times \frac{y[+i_2 + \cdots +]}{n} \times \cdots \times \frac{y[+ \cdots + i_K]}{n}$$

where, for example $\frac{y[i_1+\cdots+]}{n}$ is the fitted marginal distribution for variable A_1 and so on, and n is the total count for the table. Then, replace the CURF cell count by

$$y_s[i] = \tau y[i] + (1 - \tau)\pi^{(IND)}[i] \tag{5}$$

where $1-\tau$ is a small positive number. Note that this will obliterate the structural zeroes, but the independence component of the mixture could be modified to reintroduce them.

The idea of a mixture has a Bayesian interpretation. Suppose we assign a Dirichlet prior with positive parameters $\beta[i]$ to the cell probabilities. The prior density is

$$f(\pi) = \frac{\Gamma(\sum_i \beta[i]])}{\prod_i \Gamma(\beta[i])} \prod_i \pi[i]^{\beta[i]-1}.$$

Under this distribution, the mean of $\pi[i]$ is $\beta[i]/B$ where $B = \sum_i \beta[i]$. The variance is controlled by B; the bigger B, the smaller the variance. The posterior density is also

Dirichlet, with parameters $\beta_i + y[i]$, and the mean of the posterior density (i.e. the Bayes estimate of the cell probability) is

$$\frac{y[i] + \beta[i]}{\sum_i (y[i] + \beta[i])}.$$

If we set $\tau = \frac{n}{n+B}$ and $\beta[i] = B\pi^{(IND)}[i]$ we obtain (5). Thus, a small value for $1 - \tau$ reflects our uncertainty that the independence model is correct. The independence model is chosen purely for convenience; any other model with non-zero probabilities would suffice.

Smoothing the CURF in this way means that a log-linear model fitted to the modified data will have non-zero probabilities on all cells. This means that when we generate data from the fitted model, we will obtain records that are not included in the CURF. The same is true for mixture models; when we fit these models to the CURF data, we also obtain cell probabilities that are effectively zero for factor level combinations not present in the CURF. Thus, smoothing helps when fitting mixture models as well as log-linear models.

2.3 Mixture models

In this section we describe mixture models and discuss how their parameters may be estimated either from a single table or a set of tables each containing a subset of the variables of interest. We begin with a single table.

2.3.1 Mixture models for a single table

A mixture model for a contingency table with variables $A_1, \ldots, A_K$ represents the cell probabilities as

$$\pi[i] = \sum_{t=1}^{T} \tau_t \theta^{(1)}_{i_1 t} \theta^{(2)}_{i_2 t} \cdots \theta^{(K)}_{i_K t}$$

or, equivalently, as a mixture of independence models. For each t, the set of parameters $\{\theta^{(k)}_{i_k t}, i_k = 1, \ldots, I_k\}$ is a probability distribution on the levels $1, 2, \ldots, I_k$ of A_k, and the mixing distribution $\{\tau_t\}$ is a probability distribution on $1, 2, \ldots, T$. Mixture models are often called latent class models; the idea is that there exist a set of T unobserved latent classes, and conditional on the class, the variables $A_1, \ldots, A_K$ are independent. The number of free parameters is

$$T((I_1 - 1) + \cdots (I_K - 1)) + T - 1 = T(I_1 + \cdots I_K - K + 1) - 1.$$

For fixed T, such models can be fitted by the EM algorithm, or by solving the score equations. Suppose the data consists of a complete table of relative frequencies $f[i] = y[i]/N$, where N is the table total. Assuming a multinomial distribution for the cell counts, the log-likelihood is

$$\ell = \sum_i f[i] \log \left(\sum_{t=1}^{T} \tau_t P_{it} \right)$$

where $P_{it} = \theta^{(1)}_{i_1 t}\theta^{(2)}_{i_2 t}\cdots\theta^{(K)}_{i_K t}$. We must maximise the likelihood with respect to the parameters. Introducing a Lagrange multiplier η to account for the constraint $\sum_t \tau_t = 0$, and differentiating with respect to τ_t gives

$$\sum_i f[i]\frac{P_{it}}{\sum_{t=1}^{T}\tau_t P_{it}} = \eta \tag{6}$$

or

$$\sum_i f[i]Q[i,t] = \tau_t\eta$$

where

$$Q[i,t] = \frac{\tau_t P_{it}}{\sum_{t=1}^{T}\tau_t P_{it}}. \tag{7}$$

Adding over t gives

$$\eta = \sum_t\sum_i f[i]Q[i,t] = 1$$

since $\sum_t Q[i,t] = 1$, so that

$$\tau_t = \sum_i f[i]Q[i,t] \tag{8}$$

is an updating equation for τ_t. Similarly, differentiating with respect to $\theta^{(k)}_{lt}$ and allowing for the constraint $\sum_l \theta^{(k)}_{lt} = 1$ yields the updating equation

$$\theta^{(k)}_{lt} = \frac{\sum_{(k,l)} f[i]Q[i,t]}{\tau_t} \tag{9}$$

where $\sum_{(k,l)}$ denotes summation over all cells of the table for which $A_k = l$. This suggests that we can solve the score equations $\frac{\partial\ell}{\partial\tau_t} = 0$, $\frac{\partial\ell}{\partial\theta^{(k)}_{i_k t}} = 0$ by the following algorithm:

Step 1: Initialise $\{\theta^{(k)}_{i_k t}\}$ and $\{\tau_t\}$, by setting them equal to random distributions (say by normalizing random uniform numbers).

Step 2: Compute $D_i = \sum_t \tau_t P_{it}, i = 1,\ldots,I$.

Step 3: For $t = 1,\ldots,T$:

[3.1:] Compute $Q[i,t] = \tau_t P_{it}/D_i, i = 1,\ldots,I$.

[3.2:] Update $\tau_t = \sum_i f[i]Q[i,t]$.

[3.3:] Update $\theta^{(k)}_{lt} = \sum_{(k,l)} f[i]Q[i,t]/\tau_t$.

Step 4: Repeat Steps 2-3 until convergence.

Note that using the EM algorithm leads to exactly the same set of equations. The method does not always converge to a global maximum of the likelihood, although the likelihood increases at each step. For this reason, it is desirable to repeat the fitting, using different random starts each time, say 20 times to improve the log-likelihood, or equivalently, the degree to which the fitted distribution approximates the contingency table. To choose the value of T, the usual model selection criteria such as AIC and BIC can be used: either $AIC = G^2 + 2p$ or $BIC = G^2 + p \log n$ where p is the number of parameters.

2.3.2 Merging data sets using mixture models

Suppose that we have a population for which K variables $A_1, \ldots, A_K$ are measured. We want to describe the joint distribution of $A_1, \ldots, A_K$ using a mixture model, but have no sample from the population in which all the variables are measured. Instead, we have several samples, each containing measurements on a subset of the variables. How can we proceed? It turns out that, as long as there is a reasonable overlap between the samples, we can still fit the mixture model.

Let $\mathcal{S}_r, r = 1, \ldots, R$ be the set of variables measured in the rth sample. As in our discussion of fitting mixtures to a single population, we assume that the rth sample $\mathcal{S}_r = \{A_{j_1}, \ldots, A_{j_{\nu_r}}\}$ is represented as an array of $I_{j_1} \times \cdots \times I_{j_{\nu_r}}$ relative frequencies $f[i_1, \ldots, i_{\nu_r}]$. We denote summation over all the factor level combinations $i_1, \ldots, i_{\nu_r}$ in $\mathcal{S}_r$ by $\sum_{(r)}$. The log-likelihood for sample r is

$$\sum_{(r)} f[i_1, \ldots, i_{\nu_r}] \log \left(\sum_t \tau_t P[i_1, \ldots, i_{\nu_r}, t] \right)$$

and the log-likelihood for the complete sample is

$$\sum_{r=1}^{R} \sum_{(r)} f[i_1, \ldots, i_{\nu_r}] \log \left(\sum_t \tau_t P[i_1, \ldots, i_{\nu_r}, t] \right).$$

where now $P[i_1, \ldots, i_{\nu_r}, t] = \prod_{k=1}^{\nu_r} \theta_{i_k t}^{(j_k)}$. An argument similar to that used in the single sample case shows that the parameters of the mixture model can be estimated using the updating scheme

$$\tau_t = \frac{1}{R} \sum_{r=1}^{R} \sum_{(r)} f[i_1, \ldots, i_{\nu_r}] Q[i_1, \ldots, i_{\nu_r}, t] \tag{10}$$

and

$$\theta_{lt}^{(k)} = \frac{1}{R\tau_t} \sum_{r=1}^{R} \sum_{(k,l,r)} f[i_1, \ldots, i_{\nu_r}] Q[i_1, \ldots, i_{\nu_r}, t], \tag{11}$$

where

$$Q[i_1, \ldots, i_{\nu_r}, t] = \frac{\tau_t P[i_1, \ldots, i_{\nu_r}, t]}{\sum_t \tau_t P[i_1, \ldots, i_{\nu_r}, t]}$$

and $\sum_{(k,l,r)}$ denotes summation over all factor level combinations in $\mathcal{S}_r$ for which $A_k = l$. The sum is interpreted as zero if A_k is not in $\mathcal{S}_r$.

The algorithm described in Section 2.3.1 for a single table can be generalised to cover the case of several tables. The equations (10) and (11) lead to the algorithm

Step 1: Initialise $\{\theta_{lt}^{(k)}\}$ and $\{\tau_t\}$, by setting them equal to random distributions (say by normalizing random uniform numbers).

Step 2: For $t = 1, \ldots, T$:

[2.1] Set $S = 0$, $S_{kl} = 0,\ l = 1, \ldots, I_k,\ k = 1, \ldots, K$.

[2.2] For $r = 1, \ldots, R$

[2.2.1] Compute $D[i_1, \ldots, i_{\nu_r}] = \sum_t \tau_t P[i_1, \ldots, i_{\nu_r}, t]$.

[2.2.1] Compute $Q[i_1, \ldots, i_{\nu_r}, t] = \tau_t P[i_1, \ldots, i_{\nu_r}, t]/D[i_1, \ldots, i_{\nu_r}]$.

[2.2.1] Update $S = S + \sum_{(r)} f[i_1, \ldots, i_{\nu_r}]Q[i_1, \ldots, i_{\nu_r}, t]$.

[2.2.1] For $j = j_1, \ldots, j_{\nu_r}$, $l = 1, \ldots I_j$,

update $S_{jl} = S_{jl} + \sum_{(j,l,r)} f[i_1, \ldots, i_{\nu_r}]Q[i_1, \ldots, i_{\nu_r}, t]$.

[2.3] Set $\tau_t = S/R$ and $\theta_{lt}^{(k)} = S_{kl}/S, l = 1, \ldots, I_k,\ k = 1, \ldots, K$.

Step 3: Repeat step 2 until convergence.

An R implementation of this algorithm is discussed in Section 3.3.2.

2.3.3 Generating data from a mixture model

One of the advantages of a mixture model is the ease with which we can generate data. To generate a record, we simply

1. Draw a value of t according to the probabilities $\{\tau_t\}$;
2. For $k = 1, \ldots, K$, draw a value for A_k according to the probabilities $\{\theta_{lt}^{(k)}\}$.

2.4 Fitting models to data and probability tables

Conventionally, the models for contingency tables that we have discussed above are fitted to tables of counts which represent a sample from some population. However, in the context of this report, the counts we deal with are either come from very large samples, or compete censuses. It seems reasonable to blur the distinction between relative frequencies and probabilities in this case. The algorithms we have discussed above yield maximum likelihood estimates when either supplied with tables of counts or with relative frequencies. If we identify the relative frequencies with probabilities,

we can view the fitting of models as finding the best approximation within a class of probabilities to the table of census relative frequencies, regarded as probabilities.

For example, consider a table of relative frequencies, derived from a census (e.g. from Tablebuilder), which we regard as a table of probabilities. If we fit a mixture model with a fixed number of components to this table using maximum likelihood, assuming multinomial sampling, we are effectively finding the mixture model that best approximates (in the Kullback-Leibler sense) the census table of probabilities. That is, given a set of census probabilities $\{\pi[i]\}$, we are finding the set of probabilities $\{\pi[i]^{(MIX)}\}$ in the class of mixture models with a fixed number of components which minimises the Kullback-Leibler distance

$$KL(\pi, \pi^{(MIX)}) = \sum_i \pi[i] \log \left(\pi[i] / \pi[i]^{(MIX)} \right) .$$

It is in this sense that we regard our model fitting in the rest of this report: we are tying to represent the very high-dimensional table of census probabilities by means of a parametric family of distributions of much lower dimension, be they a log-linear family or a mixture family.

3 DATA STRUCTURES, ALGORITHMS AND SOFTWARE

In this section, we define the data structures used in our software, and discuss a set of R functions which implement the methods described in Section 2. We apply these methods to more realistic examples, and explore the limits of the log-linear and mixture approaches.

3.1 Data Structures

There are two data structures in R suitable for representing contingency tables, *arrays* and *data frames*. We discuss each in turn, and then describe some R functions for converting from one representation to the other.

3.1.1 Arrays

An array in R can have one or more dimensions, and is indexed by one or more subscripts, one per dimension. Arrays can represent a table directly, as the cell count corresponding to A_1 at level i_1, A_2 at level i_2, and A_K at level i_K, can be stored in the $(i_1, i_2, \ldots, i_k)$ position in the array. For example, the employment status-sex-labour force status data discussed in Example 1, can be stored in a $5 \times 2 \times 2$ array `my.array`, with the array element `y[i1,i2,i3]` storing the count of the (i_1, i_2, i_3) cell.

Arrays are created in R using the `array` function. We need to supply several pieces of information to this function. First, we must specify the list of factors, in some fixed order. Second, for each factor, we need to specify an ordered set of factor levels. Finally, we need to supply the vector of cell counts. Given the order of the factors and the ordering of the levels for each factor, we can establish a reverse lex order for the cells. In order to construct a "standard" table of the type described in section 2.1, the cell counts are supplied in this order.

The array function has three arguments. The first is the cell counts in the appropriate order, as detailed above. The second is the vector of dimensions $I_1, \ldots, I_K$. The third encodes the factor name and level information, in the form of an R list. The following example illustrates the procedure.

Example 6. Consider the employment status-sex-labour force status data discussed in Example 1. There are three factors, EmploymentStatus, Sex and WorkLabForceStatus, having levels "Paid Employee","Self Employed without Employees",'Employer", "Unpaid Family Worker", "Not Stated"; "Male", "Female" and "Employed Full-Time", 'Employed Part-Time". With the factors in this order, and the levels in the order shown above, the counts in reverse lex order are 572244, 122439, 86493, 9387, 21177, 425085, 40320, 29112, 9831, 12027, 75681, 20295, 3387, 6765, 5094, 223905, 30063, 10638, 13311, 10014.

The following R code creates the array and prints it:

```
counts = c(572244, 122439,  86493,   9387,  21177, 425085,  40320,
29112,   9831,  12027, 75681,  20295,   3387,   6765,   5094, 223905,
30063,  10638,  13311,  10014)
names.and.levels = list(EmploymentStatus = c("Paid Employee",
"Self Employed without Employees","Employer", "Unpaid Family Worker",
"Not Stated"),
Sex =c("Male", "Female"),
WorkLabForceStatus = c( "Employed Full-Time", "Employed Part-Time"))
my.array =  array(counts, c(5,2,2), dimnames=names.and.levels)
my.array
 , , WorkLabForceStatus = Employed Full-Time

                                Sex
EmploymentStatus                   Male Female
  Paid Employee                  572244 425085
  Self Employed without Employees 122439  40320
  Employer                        86493  29112
  Unpaid Family Worker             9387   9831
  Not Stated                      21177  12027

, , WorkLabForceStatus = Employed Part-Time

                                Sex
EmploymentStatus                   Male Female
  Paid Employee                  75681 223905
  Self Employed without Employees 20295  30063
  Employer                        3387  10638
  Unpaid Family Worker            6765  13311
  Not Stated                      5094  10014
```

Individual counts can be referred to:

```
>my.array[1,2,2]
[1] 223905
```

Apart from the level information, all that needs to be stored is the vector of counts, which has $I_1 \times I_2 \times \cdots \times I_K$ elements.

3.1.2 Data frames

A data frame is the standard data structure in R for storing data in traditional row and column form, with rows storing data on individuals, and columns storing data on variables. To represent a contingency table as a data frame, we let each row correspond to a cell of the table. One variable, `counts` say, stores the cell counts, and there are a

further K variables storing the factor level combinations. Thus, each row of the data frame has a cell count, plus the factor level combinations that identify the cell. For the employment status-sex-labour force status example, we have

```
    Count                    EmploymentStatus    Sex WorkLabForceStatus
1  572244                       Paid Employee   Male          Full-time
3  122439 Self-Employed and Without Employees   Male          Full-time
5   86493                            Employer   Male          Full-time
7    9387               Unpaid Family Worker   Male          Full-time
9   21177                          Not Stated   Male          Full-time
2  425085                       Paid Employee Female          Full-time
4   40320 Self-Employed and Without Employees Female          Full-time
6   29112                            Employer Female          Full-time
8    9831               Unpaid Family Worker Female          Full-time
10  12027                          Not Stated Female          Full-time
11  75681                       Paid Employee   Male          Part-time
13  20295 Self-Employed and Without Employees   Male          Part-time
15   3387                            Employer   Male          Part-time
17   6765               Unpaid Family Worker   Male          Part-time
19   5094                          Not Stated   Male          Part-time
12 223905                       Paid Employee Female          Part-time
14  30063 Self-Employed and Without Employees Female          Part-time
16  10638                            Employer Female          Part-time
18  13311               Unpaid Family Worker Female          Part-time
20  10014                          Not Stated Female          Part-time
```

Data frames are usually created by reading in the data from a text file. They can also be created directly from the appropriately ordered vector of counts, using the R function `expand.grid`. The following code does this.

```
counts = c(572244, 122439,  86493,   9387,  21177, 425085,  40320,
  29112,   9831,  12027, 75681,  20295,   3387,   6765,   5094, 223905,
  30063,  10638,  13311,  10014)

EmploymentStatus.levels = c("Paid Employee","Self Employed without Employees",
  "Employer", "Unpaid Family Worker", "Not Stated")
Sex.levels = c("Male", "Female")
WorkLabForceStatus.levels = c( "Employed Full-Time", "Employed Part-Time")

EmploymentStatus_Sex_WorkLabForceStatus.df = data.frame(Count = counts,
  expand.grid(EmploymentStatus=EmploymentStatus.levels,
  Sex = Sex.levels, WorkLabForceStatus = WorkLabForceStatus.levels))
```

In contrast to arrays, the identification of the factor levels with the counts is made explicitly. The advantage of this is that we do not need to include zero counts in the

data frame. This may result in substantial saving in space if the table is sparse. Unlike arrays, there is no need to have any particular ordering of the rows.

3.1.3 Standard form

Not all data frames represent tables. However, if a data frame has a single count variable, having non-negative integer values, and all the other variables are factors, then the data frame corresponds to a unique table, up to the order of the factors. We will say a data frame is in *standard form* if the count variable is the first variable, and the rows are in reverse lexicographic order.

3.1.4 Converting between formats

We have provided some R functions to convert between data frames and arrays. To convert a data frame into an array, we must restore the zero counts in the proper places. To convert from an array into a data frame in standard form, we eliminate the zero counts. There is also a function to test if a data frame represents a table, and to convert such a data frame into a data frame in standard form. The functions are described in Table 5, and are fully documented in Appendix A.1. There is also an R class "table": and some functions (`is.table`, `as.table`, `as.data.frame`) for converting. However, these do not implement the idea of a standard table. Moreover, it is possible for arrays with negative elements to be tables in the R sense, so we make no use of the R "table" class in this report.

Table 5: R functions for converting between formats.

Function name	Argument(s)	Purpose	Returns
`df2table`	A data frame	Checks if a data frame represents a table and if so converts it into an array	An array
`df2table.mix`	A data frame and a mixing proportion	Converts a data frame into an array and mixes it with a fitted independence model	An array
`as.standard`	A data frame	Checks if argument is a data frame which represents a table. If so, converts it into a data frame in standard form	A data frame in standard form.
`is.standard`	Any R object	Checks if argument is a data frame which represents a table	TRUE or FALSE
`table2df`	An array	Checks an array is non-negative and if so converts it into data frame in standard form	A data frame in standard form

3.1.5 Mixing data sets with independence models

In Section 2.2.8, we argued that fitting data to samples of real or confidentialised microdata (e.g. a CURF) and generating data from the resulting model was effectively equivalent to resampling from the CURF, and thus added little in the way of confidentiality protection. The remedy proposed was to mix the empirical distribution derived from the CURF with an independence model fitted to the CURF data. Below, we describe an R function for doing this.

To produce an array containing the mixture distribution, we can use the conversion functions above to create a table storing the empirical distribution derived from the CURF, and the R implementation of the IPF algorithm described below in Section 3.2.1 to create the table containing the fitted independence model. These are then added together in the desired proportions to produce the final result. We have written an R function `df2table.mix` to do this. It takes as its arguments the data frame containing the relevant subset of CURF variables, say `subcurf.df`, and the desired mixing proportion `tau`. It produces an array containing the complete table corresponding to the distribution

$$\pi_s[i] = \tau \frac{y[i]}{n} + (1-\tau)\pi^{(IND)}[i]$$

defined in Section 2.2.8. Thus, to create an array `mixed.table`, we could type

```
tau=0.99
mixed.table = df2table.mix(subcurf.df, tau)
```

3.2 Fitting log-linear models

In this section, we discuss fitting log-linear models to tables of probabilities, using either the IPF algorithm of Section 2.2.3 or the Fisher scoring approach described in Section 2.2.5.

3.2.1 Iterated proportional fitting

The theory of iterated proportional fitting (IPF) was described briefly in Section 2.2.3. In this section we describe an R function to implement this method. Recall that the IPF algorithm fits a hierarchical log-linear model by repeatedly adjusting the table so that its margins match a specified set of marginal tables which correspond to the log-linear model being fitted. Thus, rather than specifying the model in terms of interactions, the model is determined by a set of margins corresponding to the maximal terms in the log-linear model. We assume that the set of margins is specified in terms of a list of arrays, where each element of the list stores a particular margin in the form of an array.

Our R implementation stores the whole of the fitted table in the R workspace, so that the size of the problem it can handle is restricted. In addition, we require an amount of working storage equal to an additional table in order to achieve a reasonable trade-off between speed and size. Nevertheless, tables of up to six million cells can be handled in a reasonable amount of time, as illustrated in Example 8 below. To control

the number of iterations, our function allows the specification of a maximum number of iterations. We do not monitor the change of probabilities from iteration to iteration, as this would impose additional storage demands.

The marginal arrays of relative frequencies which serve as inputs may come from different sources, but should all be derived from the same population. For example, if we wanted to fit the model including all three-factor interactions but no higher interactions, we need to supply all the three-dimensional marginal tables. If these are available from the complete census, (for example using Tablebuilder) we would prefer these. However, if some are unavailable, we could substitute marginal tables calculated from the census CURF or any similar source.

If some of the marginal tables have sampling zeroes, we suggest smoothing these tables as described in Section 3.1.5. This will avoid restricting the support of the fitted table unnecessarily.

If the marginal tables come from different sources, or have been subject to base 3 rounding to protect confidentiality, then the submargins of one margin may be incompatible with the submargins of the other. Thus, for example, suppose we have four factors A, B, C and D, and the ABC and ABD marginal tables. If the tables come from different sources, the AB submargin of the ABC table will not necessarily match the AB submargin of the ABD table. This will affect the convergence of the IPF algorithm. However, provided the discrepancies are not too great, the table produced after a reasonable number of iterations will still match the margins with acceptable precision. Note that it is possible to use integer programming techniques to adjust the tables to remove this incompatibility (see Lee 2007, for example), but this is only practical for relatively small tables.

The R function `ipf.fitter` will take a list of marginal arrays containing relative frequencies as inputs and produce a fitted table as an output. There is also an argument `MAXITER` to control the maximum number of iterations. Note that in order for the function to keep track of the margins, it is essential that the arrays have named dimensions. This will be the case if they are created as in the examples above.

Example 7. Recall the Census data discussed in Example 3. The variables considered there were `OccupationCode`, `Sex`, `TotalHoursWrkd`, and `TotalIncomeGroup`, measured on the employed usually resident population aged 15 and over, from the 2001 Census of Population and Dwellings. Suppose we want to fit a three-interaction model, which requires the four three-dimensional marginal tables. Of these, two are available from Tablebuilder, and two from the 2001 Census CURF. We assume that we have available the four tables in the form of data frames in standard form, containing the counts. These are `hours_occupation_sex.df` and `hours_income_occupation.df`) which are derived fom the CURF) and `income_occupation_sex.df` and `income_hours_sex.df`, derived fom Tablebuilder. Details of how these data frames may be read into R are given in Chapter 4. Since our function `ipf.fitter` expects a list of relative frequency arrays as input, our first job is to create this list, by converting the data frames to arrays, calculating the relative frequencies, and assembling the arrays into a list. We type

```
# convert to an array
income_occupation_sex = df2table(income_occupation_sex.df)

# calculate relative frequencies
income_occupation_sex = income_occupation_sex/sum(income_occupation_sex)

# and repeat for the others
income_hours_sex = df2table(income_hours_sex.df)
income_hours_sex = income_hours_sex/sum(income_hours_sex)

hours_occupation_sex = df2table(hours_occupation_sex.df)
hours_occupation_sex = hours_occupation_sex/sum(hours_occupation_sex)

hours_income_occupation = df2table(hours_income_occupation.df)
hours_income_occupation = hours_income_occupation/sum(hours_income_occupation)

# make the list
array.list = list(hours_occupation_sex, hours_income_occupation,
 income_occupation_sex, income_hours_sex)
```

Finally, we calculate the fitted table of relative frequencies:

```
result.table = ipf.fitter(array.list, MAXITER=100)
```

To check that this fitted table has margins matching those of the input tables, we calculated the marginal frequencies from the fitted table, and compared them to the input tables. In each of the four plots in Figure 2, we have plotted the marginal frequencies derived from the fitted table against the corresponding frequencies from the input tables. The match is very good for the Tablebuilder tables, less so for the CURF tables, reflecting the fact that the CURF margins being based on a sample, are are less reliable than the Tablebuilder margins. Note that each cycle of the IPF algorithm adjusts the fitted table to match each margin in turn, the order of adjustment being the order of the margins in the list. For this reason it is advisable to put the CURF margins at the beginning of the list.

Example 8. To explore the limits of IPF technique, we constructed the series of test problems shown in Table 6. All problems ran successfully. The timings shown are in seconds, using a maximum of 20 iterations. The speeds could be increased by coding the IPF routine in FORTRAN or C but seem acceptable coded in R.

3.2.2 Fitting log-linear models using Fisher scoring

Fisher scoring is the standard method for fitting log-linear models. In particular, it is the method used by the R function `glm`. However, this standard R implementation is not designed for fitting large tables, as both the vector of counts and the X matrix are explicitly formed and stored in the R workspace. Instead, we have written a function

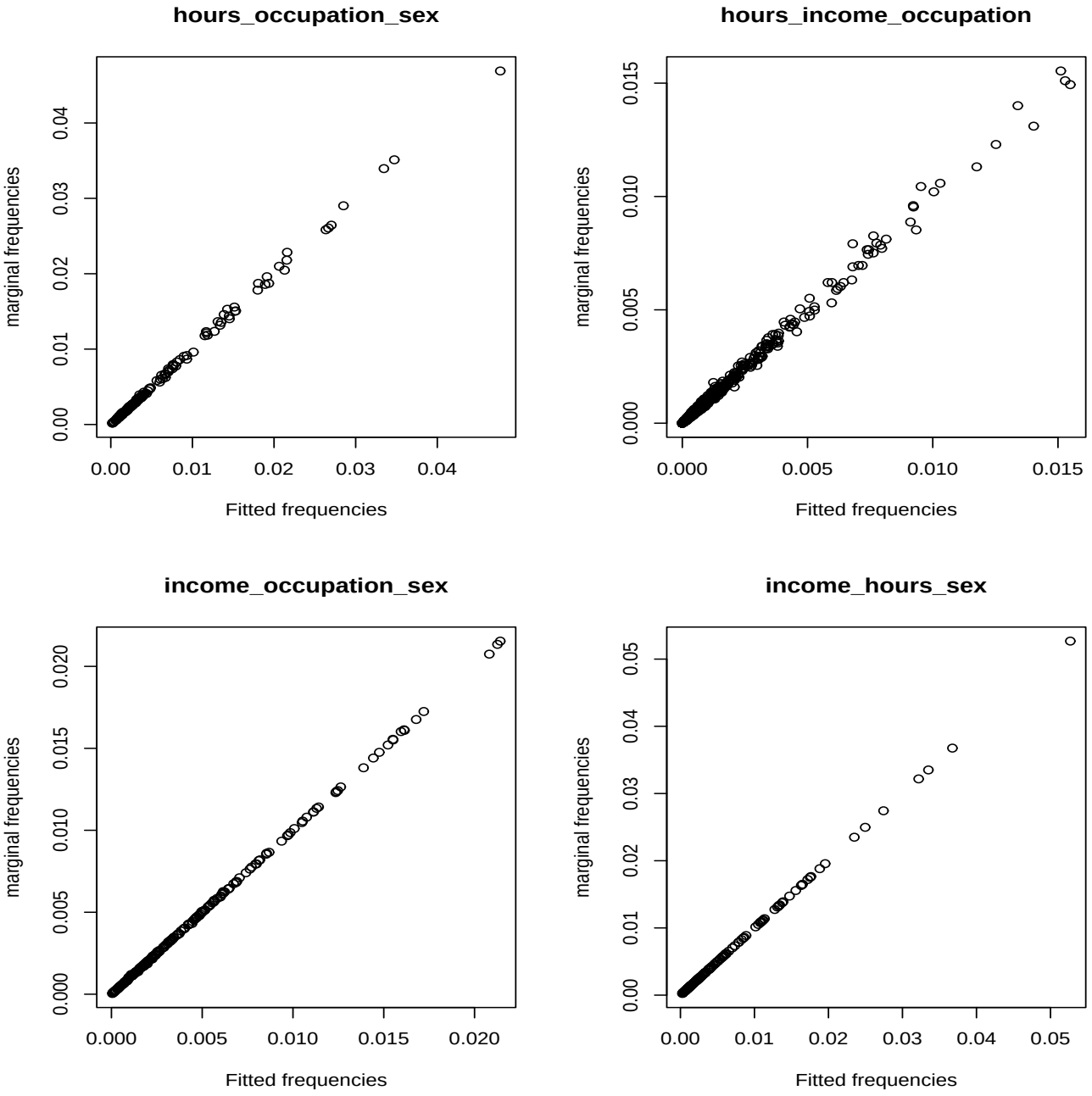

Figure 2: Fitted and actual margins for Example 7.

Table 6: Results of running `ipf.fitter`

Number of factors (K)	Dimensions $(I_1, \ldots, I_K)$	Number of cells	Number of parameters	Margins	Time (Seconds)
3	3,4,2	24	18	All 2-dim	0.05
5	3,4,2,4,6	576	89	All 2-dim	0.11
5	3,4,2,4,6	576	273	All 2-dim	0.28
6	2,3,5,6,2,5	1800	1640	All 5-dim	0.56
7	10,3,5,3,9,5,8	162,000	568	All 2-dim	8.01
7	10,3,5,3,9,5,8	162,000	4708	All 3-dim	16.34
7	10,3,5,3,9,5,8	162,000	129,774	All 6-dim	52.19
8	10,3,5,3,9,5,8,6	972,000	735	All 2-dim	79.08
8	10,3,5,3,9,5,8,6	972,000	7548	All 3-dim	150.23
9	10,3,5,3,9,5,8,6,2	1,944,000	765	All 2-dim	195.79
9	10,3,5,3,9,5,8,6,6	5,832,000	963	All 2-dim	591.34

`log.lin.fitter` which follows the approach outlined in Section 2.2.2 and does not require the whole table to be stored in memory, but reads blocks of counts from a disk file as required. To avoid the requirement of storing X, we explicitly form its rows as each block of counts is read in. This requires that the counts are in reverse lexicographic order. Since Fisher scoring puts limits on the number of parameters rather than the number of cells, out function will fit models having all interactions above second order set to zero. We assume that the counts in reverse lexicographic order are stored in a disk file, with each line of the file (containing multiple counts) forming a block of counts. To calculate the Fisher scoring updates, the counts are read in block by block to form the score vector and information matrix. The following example illustrates the use of the function `log.lin.fitter`. Note that the starting values for the Fisher scoring iterations are provided by fitting a linear model using the log counts as a response.

Example 9. We constructed a test example, by randomly generating a table corresponding to five factors A, B, C, D and E with 2,3,4,3,6 levels respectively. The resulting 432 cell counts were generated by independently sampling from a Poisson distribution with mean 10, and stored in reverse lex order in a text file `testfile.txt`. The R code to do this is

```
test.df = data.frame(Count=rpois(prod(Kvec),10), expand.grid(A=factor(1:2),
B=factor(1:3), C=factor(1:4),D=factor(1:3),E=factor(1:6)))
countfile = "testfile.txt"
write(test.df$Count, countfile, ncolumns=100)
```

The resulting file has 432 entries, with 100 on each line except the last. To fit the model, we type

```
Kvec=c(2,3,4,3,6)
beta = log.lin.fitter(countfile, Kvec)
```

This example is small enough to use the R `glm` function to fit the model. The code below re-fits the model using `glm` and plots the coefficients obtained using `glm` versus

those obtained above using `log.lin.fitter`. As we see from Figure 3, the agreement is excellent.

```
beta.glm = coef(glm(count~(A+B+C+D+E)^2, family=poisson, data=test.df))
plot(beta,beta.glm, xlab = "Coefficients from log.lin.fitter",
   ylab="coefficients from glm")
```

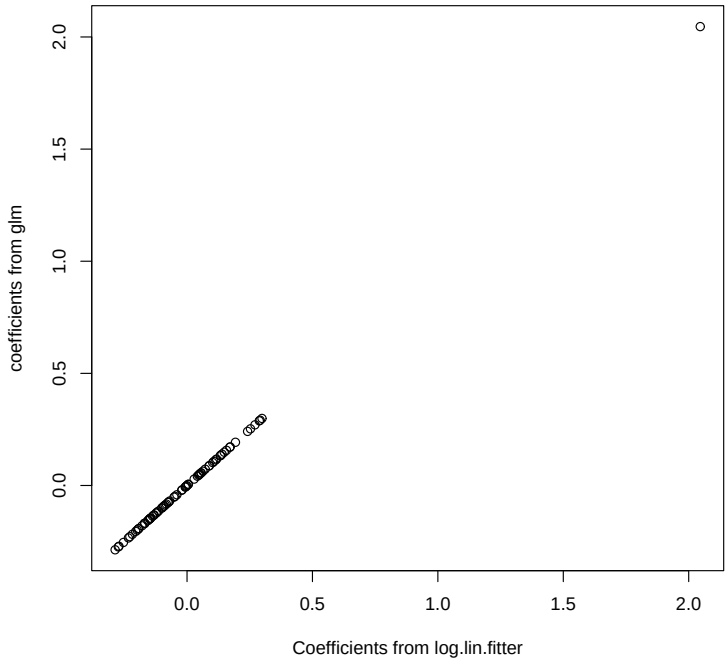

Figure 3: Coefficients calculated by two methods for Example 9.

3.3 Fitting mixture models

In this section, we describe a set of R functions to fit mixture models and illustrate their use. We discuss both the case of fitting a mixture to a single table, and also the use of mixtures to fit a model to several marginal tables.

3.3.1 Fitting single tables

First, we consider the situation where we fit a mixture model to a single table. Such a situation might arise when we have a sample such as a CURF, and want to fit a mixture to a complete table obtained by smoothing the empirical table. Our implementation, written in R, has the same storage requirements as the IPF function described above.

Example 10. We illustrate using the variables `OccupationCode`, `Sex`, `TotalHoursWrkd`, and `TotalIncomeGroup` considered in Example 7. We assume that we have available

a data frame in standard form, representing the table derived from the 2001 census CURF. We first smooth it, producing a smoothed complete table of relative frequencies, and then fit the mixture model using the function `mixture.fitter.single`, which takes as arguments a single table in the form of an array, and the number of mixture components T. The function returns the parameters of the mixture model in the form of a list with two components `tau` and `theta`. The first list element `tau` is the vector of mixture probabilities, and the second `theta` is a list of K matrices, where the kth matrix is an $I_k \times T$ matrix whose tth column contains the distribution of A_K for the tth mixture component.

To mitigate the problem of landing in a local minimum, we use 20 randomly chosen starting values, fit the model each time, and record the best fit to the smoothed table, as measured by the Kullback-Leibler distance. We took $T = 10$. The R code is

```
# create mixed array
mixed.array= df2table.mix(hours_income_occupation_sex.df, tau=0.99)

# now fit model 20 times, choose best fit
T=10
params = mixture.fitter.single(mixed.array, T)
KL.best=KLDist(mixed,array,params)
for(i in 1:20){
params = mixture.fitter.single(mixed.array, T)
KL.current=KLDist(mixed.array,params)
if(KL.current<KL.best){
params.best = params
KL.best = KL.current
}}
```

After the execution of this code, the parameters of the best fitting model are contained in the object `params.best`. As in Example 7, we can compare the three-dimensional margins of the input table with the three-dimensional margins of the fitted model. These are shown in Figure 4, plotted on a square-root scale. The fit appears satisfactory.

Example 11. To investigate the performance of the mixture method, we used the same series of test examples as those used in Example 6. The results are shown in Table 7. Again, all problems ran successfully. The timings shown are in seconds, using a maximum of 20 iterations as in Example 6, and a single fit of the model each time. As one would expect from the algorithm, the timings are approximately linear in T. The mixture model is much slower to fit than the log-linear, so a case can be made for coding it in a lower-level language. This is a future project.

3.3.2 Fusing data sets

As described in Section 2.3.2, the basic algorithm used to fit a single mixture model can be adapted to fit a model to several tables, each containing data on a subset of

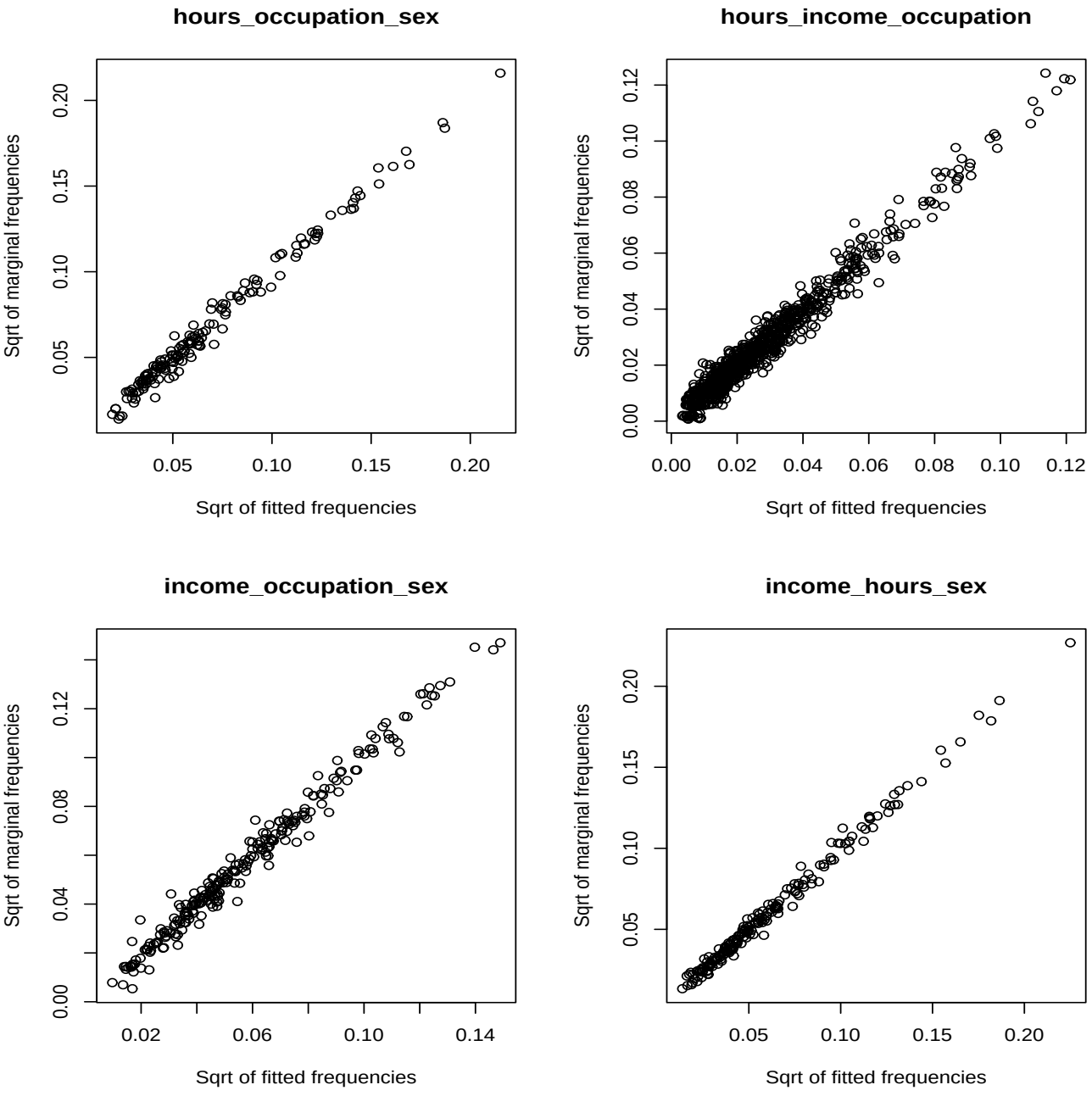

Figure 4: Fitted and actual margins for Example 10.

Table 7: Results of running `mixture.fitter.single`

Number of factors (K)	Dimensions $(I_1, \ldots, I_K)$	Number of cells	Number of parameters	T	Time (Seconds)
3	3,4,2	24	20	3	0.19
5	3,4,2,4,6	576	74	5	0.68
5	3,4,2,4,6	576	149	10	1.30
5	3,4,2,4,6	576	224	15	1.78
6	2,3,5,5,2,5	1,800	89	5	1.60
6	2,3,5,5,2,5	1,800	179	10	3.07
6	2,3,5,5,2,5	1,800	269	15	4.52
7	10,3,5,3,9,5,8	162,000	184	5	8.01
7	10,3,5,3,9,5,8	162,000	369	10	16.34
7	10,3,5,3,9,5,8	162,000	554	15	52.19
8	10,3,5,3,9,5,8,6	972,000	209	5	344.89
8	10,3,5,3,9,5,8,6	972,000	419	10	689.45
8	10,3,5,3,9,5,8,6	972,000	629	15	1043.92
9	10,3,5,3,9,5,8,6,2	1,944,000	214	5	816.77
9	10,3,5,3,9,5,8,6,6	5,832,000	234	5	2806.84

variables. The R function `mixture.fitter.fusion` implements this algorithm. Suppose we have a set of tables, each a sample from the same population, but containing different variables. The function assumes that each sample is in the form of an array, and the arrays representing the samples have been combined as a list. Note that it is essential that the a variable appearing in several tables must have the same name and factor levels in each table. The function `mixture.fitter.fusion` takes this list as its argument and returns the parameters of the fitted mixture distribution. The other input parameters are the same as those in the function `mixture.fitter.single` used for a single mixture.

Example 12. To illustrate the use of `mixture.fitter.fusion`, we revisit the data sets discussed in Example 2. In that example, we had two data sets from the 2001 census, one involving the variables “Sex” “TotalHrsWrkd”, and “TotalIncomeGroup” (in the data frame `income_hours_sex.df`) and the other the variables “OccupationCode”, “Sex” and “TotalIncomeGroup”, in the data frame `income_occupation_sex.df`.

To fit the model, we first need to convert the data frames to arrays, and form the arrays into a list. The code is

```
A = df2table(income_hours_sex.df)
A = A/sum(A) # convert to relative frequencies
B = df2table(income_occupation_sex.df)
B = B/sum(B)# convert to relative frequencies
array.list = list(A,B)
```

To fit a model with say 10 components, we type

```
params.both=mixture.fitter.fusion(array.list, T=10, MAXITER=100)
```

The result `params.both`, is a list with three components as for the case of fitting a single table: the mixture probabilities τ_t, the parameters of the mixture distributions as a list of matrices, and list containing the names and levels of the variables. This code was in fact the code used to produce Figure 1 in Section 1.4.

3.4 Generating synthetic data sets

The functions we have described so far fit probability distributions to data sets derived from a variety of sources. These distributions are represented in various ways, either explicitly as a probability table in the form of an array, or implicitly as a set of parameters defining a log-linear or mixture model. In this section, we describe a set of functions that can be used to generate synthetic data sets from the fitted probability distributions. The data sets produced are csv files that can be manipulated using tools such as Microsoft Excel.

3.4.1 Generating data from an explicit table

This is the simplest method, and uses the inversion method described in Section 2.2.6. The R function `generate.sample.prob` takes as its input a probability table $\{\pi[i]\}$ in the form of an array and produces a csv file of a specified number N of records of the form $i = (1_i, \ldots, i_K)$ where each record is a random draw from the distribution $\{\pi[i]\}$.

Example 13. Consider the four-variable example discussed in Example 7. In the code given there, a distribution was fitted and the fitted probability table was stored in the array `result.table`. To generate a csv file `example7.csv` of say 50,000 records using this table, we can type

```
file = "example7.csv"
N = 50000
generate.sample.prob(N, file, result.table)
```

This will create a csv file `example7.csv` in the working directory.

3.4.2 Generating data from a log-linear parameter vector

In this case we have two options. If the probability table will fit into memory, we can calculate the table from the parameter vector using the function `loglin2table`. This takes three arguments; the parameter vector as calculated by the function `log.lin.fitter`, the vector containing the number of levels of the various factors (this should be identical to that used as input to `log.lin.fitter`), and a list containing the dimension names for the output array. Care should be taken with the inputs, as there is no information about the factors stored in the log-linear parameter vector. The function produces a probability table that can be used as input to the function `generate.sample.prob` described above. The following example illustrates the procedure:

Example 14. In Example 9, we considered an artificial example with five variables, A, B, C,D and E, with numbers of levels 2,3,4,3,6 respectively. A complete probability

table was constructed and stored as a file, using the reverse lexicographic order implied by this alphabetic ordering of the factors: i.e. the entries are arranged in the input file so that the values of A vary most rapidly, those of B next most rapidly, and so on. The vector `Kvec` is thus `c(2,3,4,3,6)`. The log-linear model was fitted, producing a vector `beta`. The following code calculates the probability table as the array `result.table`.

```
Kvec = c(2,3,4,3,6)
dimnames = list(A=1:Kvec[1], B=1:Kvec[2], C=1:Kvec[3], D=1:Kvec[4],
        E=1:Kvec[5])
result.table = loglin2table(beta, Kvec, dimnames)
```

We then proceed as in Example 13.

If the probability table is too large to fit in memory, we can use the Metropolis-Hastings method described in Section 2.2.7 to generate random draws from the probability table defined by the log-linear parameters. This required that we have available the one-dimensional marginal distributions of each of the factors in the log-linear model. These are represented as a list of vectors, each vector containing the marginal probabilities of the corresponding factor. The function `make.margin` will construct this list from the same inputs as those used in `log.lin.fitter`. Alternatively, if the table will fit into memory, we can use the standard R functions to construct the list, as shown in the example below. Again, care is necessary in the construction of this list: the order of the factors in the list must match the order used to construct the parameter vector, and the order of the probabilities within each vector must also be consistent with the ordering in the parameter vector. The example below illustrates how to ensure this, using the same data as in the example above.

Example 15. Consider again Example 9 with five variables A, B, C,D and E, with numbers of levels 2,3,4,3,6 respectively. The relative frequencies used to fit the log-linear model are stored in the file `testfile.txt` in reverse lex order. We can construct the margin list by the code

```
Kvec = c(2,3,4,3,6)
margin.list = make.margin("testfile.txt", Kvec)
```

Alternatively, if the relative frequencies were stored in the array `freq`, we could type

```
Kvec = c(2,3,4,3,6)
K = length(Kvec)
margin.list = vector(length=K, mode="list")
for(k in 1:K) margin.list[[k]] = apply(freq, k, sum)
```

Having created the margin list, the following code will generate 50,000 records in the csv file `MH.csv` from the distribution defined by the log-linear model with parameter vector `beta`:

```
file = "MH.csv"
N=50000
generate.sample.MH(N, file, beta, margin.list, thin=10, burn = 1000)
```

3.4.3 Generating data from a mixture model

To draw a set of N records from a mixture model, we first draw a single random vector $N_1, \ldots, N_T$ from a multinomial distribution with parameters N and $\{\tau_t\}$. Then, for $t = 1, \ldots, T$, we draw N_t values of the form $(A_1, \ldots, A_K)$, where the values of A_k are drawn independently from the distribution $\{\theta_{lt}^k\}$. This simple algorithm is implemented by the R function `generate sample.mix`.

Example 16. Consider the four-variable example discussed in Examples 5 and 10. A mixture model was fitted and a set of parameters `params.best` was estimated. To generate 50,000 records from the distribution specified by this set of parameters, and store the result in a file `mix.csv`, type

```
N=50000
file = "mix.csv"
generate.sample.mix(N, file, params.best)
```

4 APPLICATION TO CENSUS DATA

In this section of the report, we apply the methods discussed in the previous sections to data from the 2001 Census of Population and Dwellings. We first describe a set of marginal tables, derived from the Statistics New Zealand web site. We cover the structure of the tables, the variables they contain, the populations they refer to, and the way tables can be read into R. Next, we discuss how the CURF can be used to create tables similar to those on the web site, thus filling any gaps that remain. Finally, we apply our methods to these data and illustrate how the creation of synthetic data sets proceeds.

4.1 Marginal tables

In this section, we describe a set of marginal tables created using the Tablebuilder tool on the Statistics New Zealand web site. The tables have been converted into a set of R data frames, using consistent variable names and levels which match the 2001 census CURF. This has involved some recoding of the original variables, and the dropping of some when the CURF variables could not be reconciled with those in the tables, primarily because of the differences in the treatment of ethnicity and religion. We begin with a discussion of the set of census variables we use.

In Table 8, we list the variables we consider, which are restricted to a subset of those in the CURF. This table is adapted from the CURF documentation, and shows the names of the variable, the levels of the variable, and the population it refers to. Because we want the definitions to be compatible with the CURF, we omit variables relating to ethnicity and religion. This is due to the fact that in the Tablebuilder tables, the categories for the different ethnicities and religions are not exclusive, in that one can be a member of more than one ethnic group. This makes matching CURF information to the tables impossible.

The CURF has many missing values, for questions that are not relevant to the individual concerned, for example employment for persons under 5. Rather than introduce large numbers of structural zeroes into our tables, we subset the CURF. The Tablebuilder tables we use refer to one of three populations, namely the usually resident population, the usually resident population aged 15 and over, and the employed usually resident population aged 15 and over. Subsetting the CURF along these lines largely avoids the problem of structural zeros, except for the variable “usual residence 5 years ago” which when cross-classified with age introduces structural zeroes (all the 0-4 age group is in the category “not born 5 years ago”). We dealt with this by omitting this variable from consideration.

Table 8. Description of the variables in the census tables.

Variable	Description	Levels	Population
AgeGroup	Age of individual in years grouped	0-4 Years 5-14 Years 15-19 Years 20-24 Years 25-34 Years 35-44 Years 45-54 Years 55-64 Years 65-74 Years 75 and Over	Usually Resident
AgeGroup	Age of individual in years grouped	15-19 Years 20-24 Years 25-34 Years 35-44 Years 45-54 Years 55-64 Years 65 Years and Over	Usually Resident 15+
BirthPlace	Birthplace (Country of Birth)	New Zealand Other Oceania and Antarctica Europe Asia Other Not Elsewhere Included	Usually Resident
EmploymentStatus	Status in employment	Paid Employee Employer Self-Employed and Without Employees Unpaid Family Worker Not Stated	Employed Usually Resident 15+
HighestQual	Highest qualification	No Qualification Fifth Form Qualification Sixth Form Qualification Higher School Qualification Other NZ Secondary School Qualification and Overseas Secondary School Qualification Basic Vocational Qualification Skilled Vocational Qualification Intermediate Vocational Qualification Advanced Vocational Qualification Bachelor Degree Higher Degree Not Elsewhere Included	Usually Resident 15+
IndustryGroup	Industry, grouped	Agriculture, Forestry and Fishing Manufacturing Construction Wholesale Trade, Retail Trade, Accommodation, Cafes and Restaurants Transport and Storage, Communication Services Finance and Insurance and Property and Business Services Education Health and Community Services Other Industries Not Elsewhere Included	Employed Usually Resident 15+
LegalMaritalStatus	Legal marital status	Never Married Married (Not Separated) Separated Divorced Widowed Not Elsewhere Included	Usually Resident 15+

Table 8 (Cont). Description of the variables in the census tables.

Variable	Description	Levels	Population
NumLanguagesSpoken	Languages spoken, partially grouped	None One Language Two Languages Three or more Not Elsewhere Included	usually Resident
OccupationCode	Occupation, grouped	Elementary Occupations Legislators Administrators and Managers Professionals Technicians and Associate Professionals Clerks Service and Sales Workers griculture and Fishery Workers Trades Workers Plant and Machine Operators and Assemblers Not Elsewhere Included	Employed Usually Resident 15+
SectorOfOwnership	Sector of ownership, grouped	Central and Local Government Private Not Stated	Employed Usually Resident 15+
Sex	Sex of individual	Male Female	Usually Resident
SocialMaritalStatus	Social marital status	Partnered Non-partnered Not stated	Usually resident 15+
TotalHrsWrkd	Hours worked in employment (per week) grouped	1-9 Hours Worked 10-19 Hours Worked 20-29 Hours Worked 30-39 Hours Worked 40-49 Hours Worked 50-59 Hours Worked 60-69 Hours Worked 70 Hours or More Worked Not Elsewhere Included	Employed usually resident 15+,
TotalIncomeGroup	Total personal income, partially grouped	Loss or Zero Income $1 - $5,000 $5,001 - $10,000 $10,001 - $15,000 $15,001 - $20,000 $20,001 - $25,000 $25,001 - $30,000 $30,001 - $40,000 $40,001 - $50,000 $50,001 - $70,000 $70,001 or More Not Stated	Usually Resident 15+
TravelToWorkGroup	Main means of travel to work, partially grouped	Worked at Home Did Not Go To Work Today Drove a Private Car, Truck or Van Drove a Company Car, Truck or Van Passenger in a Car, Truck, Van or Company Bus Public Bus or Train Bicycle, Walked or Jogged Motor Cycle, Power Cycle or Other Not Stated	Employed usually resident 15+
WorkLabForceStatus	Work and labour force status	Employed Full-time Employed Part-time Unemployed Not in the Labour Force Work and Labour Force Status Unidentifiable	Usually resident 15+

Table 8 (Cont). Description of the variables in the census tables.

Variable	Description	Levels	Population
YearsAtRes	Years at usual residence, partially grouped	Less than One Year 1 Year 2 Years 3 Years 4 Years 5-9 Years 10-14 Years 15-19 Years 20-24 Years 25-29 Years 30 Years or More Not Elsewhere Included	Usually resident

We also omitted variables for which there were no tables on Tablebuilder which related to the three populations described above. This left the list shown in Table 8. The list of tables we have extracted from table builder is shown in Table 9. They are grouped by the three subpopulations. These are defined as follows: the usually resident subpopulation in the consists of all persons whose residence is in New Zealand. The usually resident population 15+ consists of all persons in the usually resident subpopulation who are aged 15 or more, and the employed usually resident population 15+ consists of all persons in the usually resident subpopulation who are aged 15 or more and are employed, either full-time or part-time.

4.2 Creating census tables in R

In this section, we describe how to create the census tables as R data frames. These are constructed from two sources; the Tablebuilder tables and the census CURF.

4.2.1 Creating census tables from Tablebuilder

We have downloaded a set of tables from Tablebuilder, as a set of csv files, and written a series of R scripts to read them into the R workspace as data frames. In the software supplied, there is a folder `SNZTables` which contains three subdirectories, `usually resident`, `usually resident 15+` and `employed usually resident 15+`. Each contains a separate subfolder for each table, consisting of a csv file containing the data, and a file containing an R script which reads the csv file, defines the levels of the variables and creates a data frame. Thus, for example, in the `usually resident` folder there are three subdirectories `birthplace.sex`, `languages.age.sex`, and `yearsatres.age.sex`. The `languages.age.sex` folder contains two files, namely `languages.age.sex.csv` and `languages.age.sex.r` containing an R script to create `languages_age_sex.df`, a data frame in standard form. The data frame can be created by first setting the R working directory to `usually resident` and then typing

```
source("languages.age.sex\\languages.age.sex.r")
```

To create the entire set of data frames, use the function `make.all`. To use this function, set the R working directory to `SNZTables` and type

Table 9: List of Census Tables (by population).

Data frame	Table
Census Usually Resident Population	
birthplace_sex.df	Birthplace and Sex
languages_age_sex.df	Number of Languages Spoken, Age Group and Sex
yearsatres_age_sex.df	Years at Usual Residence, Age Group and Sex
Census Usually Resident Population Aged 15 Years and Over	
income_age_sex.df	Total Personal Income, Age Group and Sex
income_qual_age.df	Total Personal Income, Highest Qualification and Age Group
income_qual_sex.df	Total Personal Income, Highest Qualification and Sex
income_work_age.df	Total Personal Income, Work Status and Age Group
income_work_sex.df	Total Personal Income, Work Status and Sex
legalms_age_sex.df	Legal Marital Status, Age Group and Sex
qual_age_sex.df	Highest Qualification, Age Group and Sex
qual_birthplace_sex.df	Highest Qualification, Birthplace and Sex
qual_income_sex.df	Highest Qualification, Total Personal Income and Sex
qual_work_sex.df	Highest Qualification, Work Status and Sex
socialms_age_sex.df	Social Marital Status, Age Group and Sex
work_age_sex.df	Work Status, Age Group and Sex
work_income_sex.df	Work Status, Income and Sex
work_qual_sex.df	Work Status, Highest Qualification and Sex
Employed Census Usually Resident Population Aged 15 Years and Over	
emp_work_income_sex.df	Status in Employment, Work and Labour Force Status, Total Personal Income and Sex
emp_work_qual_sex.df	Status in Employment, Work and Labour Force Status, Highest Qualification and Sex
income_emp_age.df	Total Personal Income, Status in Employment and Age Group
income_emp_sex.df	Total Personal Income, Status in Employment and Sex
income_hours_sex.df	Total Personal Income, Hours Worked in Employment Per Week and Sex
income_industry_sex.df	Total Personal Income, Industry and Sex
income_occupation_sex.df	Total Personal Income, Occupation Code and Sex
industry_emp_sex.df	Industry, Status in Employment and Sex
industry_work_age.df	Industry, Work and Labour Force Status and Age Group
industry_work_sex.df	Industry, Work and Labour Force Status and Sex
occupation_age_sex.df	Occupation Code, Age Group and Sex
occupation_emp_sex.df	Occupation Code, Employment Status and Sex
occupation_industry_sex.df	Occupation Code, Industry and Sex
occupation_work_age.df	Occupation Code, Work Status and Labour Force Status and Age Group
occupation_work_sex.df	Occupation Code, Work Status and Labour Force Status and Sex
qual_emp_age.df	Highest Qualification, Status in Employment and Age Group
qual_emp_sex.df	Highest Qualification, Status in Employment and Sex
qual_industry_sex.df	Highest Qualification, Industry and Sex
qual_occupation_sex.df	Highest Qualification, Occupation and Sex
sector_age_sex.df	Sector of Ownership (Employer), Age Group and Sex
sector_income_sex.df	Sector of Ownership (Employer), Total Personal Income and Sex
sector_qual_sex.df	Sector of Ownership (Employer), Highest Qualification and Sex
sector_work_sex.df	Sector of Ownership (Employer), Work Status and Sex
travel_income.df	Main Means of Travel to Work and Total Personal Income
travel_sex.df	Main Means of Travel to Work and Sex
travel_work_emp.df	Main Means of Travel to Work, Work and labour Force Status and Status in Employment
travel_work_industry.df	Main Means of Travel to Work, Work and Labour Force Status and Industry
travel_work_occupation.df	Main Means of Travel to Work, Work and Labour Force Status and Occupation

```
make.all()
```

This will create all the data frames in the R work space. As an alternative to setting the directory, the path of the folder `SNZTables` can be specified as a path, e.g.:

```
make.all("F:\\Stats NZ\\2007 project\\SNZTables")
```

Note that the path must be specified using the R double `\\` notation, not the Windows / notation.

Many of the functions written require tables of relative frequencies as arguments, so the data frames storing the tables need to be converted to arrays first. This can be done using the function `df2table` described in Chapter 3.

4.2.2 Making data frames containing subsets of the CURF variables

In this section, we discuss the code used to construct R data frames from the csv version of the 2001 census CURF. A feature of the CURF is that not all variables are measured on each individual. For example, unemployed persons do not have a value for the variable WorkLabForceStatus, and infants do not have a value for OccupationCode.

We deal with this problem by constructing three separate data frames corresponding to the usually resident population, the usually resident population aged 15 and over, and the employed usually resident population aged 15 and over. The data frame `usually.resident.df` is constructed by selecting from the CURF all records whose value for the variable IndividualRecTypeCode is 3 or 4 (i.e. “New Zealand Child” or “New Zealand Adult”.) For these individuals, the five variables AgeGroup, BirthPlace, NumLanguagesSpoken, Sex and YearsAtRes are the only ones to have no missing values, so we retain only these variables. The CURF file is read into R, the cases and variables selected as described above, and the data massaged into a data frame in standard form, as described in Section 3.1.3. The file `usually.resident.r` contains the script for doing this. Setting the current R directory to point to the supplied folder `CURF` (this folder also contains the csv version of the CURF) and typing

```
source("usually.resident.r")
```

will cause the data frame `usually.resident.df` to appear in the R workspace. Alternatively, select “Source R code ...” from the R file menu and navigate to the file ”usually.resident.r”. The data frame `usually.resident.df` has 3,068 lines and six variables (the five above plus the table counts in the first column). This represents data on 74,767 individuals. The complete probability table for these five variables has 7,200 cells.

The second data frame `usually.resident.15plus.df` contains ten variables measured on the usually resident population aged 15 and over (the individuals in the first data frame who do not have AgeGroup equal to “0-4 Years” or “5-14 Years”. The additional five variables are HighestQual, SocialMaritalStatus, TenureHolderCode , TotalIncomeGroup and WorkLabForceStatus. The data frame is created by running the

R script in the file `usually.resident.15+.r`. It has 35,882 rows and 11 variables (ten plus the counts) and contains data on 57,821 individuals. The complete probability table for the 10 variables measured on this population has 48,988,800 cells, too many to fit into the R workspace.

The final data frame `employed.usually.resident.15plus.df` contains 16 variables measured on the employed usually resident population aged 15 and over (the individuals in the second data frame who have WorkLabForceStatus equal to "Employed Full-Time" or "Employed Part-Time". The additional variables for this data frame are EmploymentStatus, IndustryGroup, OccupationCode , SectorOfOwnership , TotalHrsWrkd and TravelToWorkGroup. The R script to create this data frame is in the file `employed.usually.resident.15+.r`. The resulting data frame has 34,056 rows and 17 variables (16 plus the counts) and contains data on 34,492 individuals. The complete probability table for the 16 variables measured on the employed usually resident population 15 and over has a whopping 2,821,754,880,000 cells!

Sometimes we want to create a marginal table from one of these three data frames. We can do this by selecting the appropriate columns. For example, suppose we want to create a table of relative frequencies for the variables AgeGroup, Sex and YearsAtRes from the data frame `usually.resident.df`. These variables are in the second, fifth and sixth columns, with the cell counts in the first:

```
> names(usually.resident.df)
[1] "Count"              "AgeGroup"           "BirthPlace"
[4] "NumLanguagesSpoken" "Sex"                "YearsAtRes"
```

We can select these columns, along with the counts, but there will be many repeat rows for each combination of the three factors in the resulting data frames. We need to compact the data frame so that each combination appears only once, and the counts are accumulated in the appropriate way. This is done with the function `compact.data.frame`. The code

```
age_sex_years.df = compact.data.frame(usually.resident.df[,c(1,2,5,6)])
```

performs this task and creates the desired data frame. We can create a table of relative frequencies for the age-sex-years margin by typing

```
temp = df2table(age_sex_years.df)
age_sex_years = temp/sum(temp)
```

4.3 Generating synthetic data using the census tables

In this section, we use the functions described in Section 3 to generate a variety of synthetic data sets from the Tablebuilder tables and the census CURF. We deal with our three subpopulations separately, beginning with the usually resident population.

4.3.1 The usually resident population

There are five variables measured on the usually resident population, and the probability table for the joint distribution of these variables has 7,200 cells. For such a small table, any of the methods we have described are appropriate. For example, we could mix the empirical table of relative frequencies with an independence table and sample from the mixed table. Alternatively, we could fit a mixed or log-linear model to achieve more drastic smoothing of the empirical frequencies. Below we give some code samples that accomplish these tasks.

First, suppose we want to create a synthetic data set of 100,000 records using the independence method, and store the result in a csv file `usually.resident.ind.csv`. Assuming the data frame `usually resident.df` has been created as described in Section 4.2.2, the following code will create the csv file:

```
prob.table = df2table.mix(usually.resident.df)
N = 100000
file ="usually.resident.ind.csv"
generate.sample.prob(N, file, prob.table)
```

To produce a similar file by fitting a mixture model, we use the code

```
freq.table = df2table.mix(usually.resident.df)
params = mixture.fitter.single(freq.table, T=10)
N = 100000
file ="usually.resident.mixture.csv"
generate.sample.mix(N, file, prob.table)
```

Finally, for the log-linear method using IPF, we need to construct a list of marginal arrays, using an array from the Tablebuilder collection if one is available, otherwise one derived from the CURF. Suppose we want to fit a model using all three-dimensional margins, equivalent to a model where all 4- and 5-order interactions are set to zero. There are 10 three-dimensional margins, so need to construct the list in a systematic way. The easiest way to do this is to use two loops to make all the tables from the CURF, and then substitute the Tablebuilder tables. The following code produces the list of arrays:

```
# create list, set elements to NULL
array.list = vector(length=10, mode="list")
# now loop
index = 1
for(i in 2:5){
for(j in (i+1):6){
temp = df2table(usually.resident.df[,-c(i,j)])
array.list[[index]] = temp/sum(temp)
index = index + 1
}
}
```

Note that each time we go through the loop, we delete columns i and j from the data frame `usually.resident.df` and convert the result into a table of relative frequencies. The tables `languages_age_sex.df` and `yearsatres_age_sex.df` are elements 7 and 5 on this list. These tables are available in our Tablebuilder collection so we substitute them:

```
temp = df2table(languages_age_sex.df)
array.list[[7]] = temp/sum(temp)
temp = df2table(yearsatres_age_sex.df)
array.list[[5]] = temp/sum(temp)
```

Now we can fit the model and generate the synthetic data:

```
prob.table = ipf.fitter(array.list)
N = 100000
file ="usually.resident.loglin.csv"
generate.sample.prob(N, file, prob.table)
```

4.3.2 The usually resident population ages 15 and over

The data frame `usually.resident.15plus.df` has eleven variables (with the variable Count in the first column) and 35,882 rows. The complete probability has 48,988,800 cells, which is too many to fit into the R workspace. To fit a model to these variables, we can adopt a fusion approach. The probability table for the first eight variables ("AgeGroup", "BirthPlace","HighestQual", "LegalMaritalStatus", "NumLanguagesSpoken", 'Sex", "SocialMaritalStatus","TotalIncomeGroup") has 816,480 cells, while the table for the last eight ("HighestQual", "LegalMaritalStatus","NumLanguagesSpoken", "Sex", "SocialMaritalStatus", "TotalIncomeGroup", "WorkLabForceStatus", "YearsAtRes") has 1,166,400 cells. To fit a model to the complete table we can adopt a hybrid approach: we can use IPF to fit the two separate tables (corresponding to the first eight and last eight variables respectively) and then use the mixture approach to fuse the two tables together, as shown in the following code fragment:

```
# make a list to store the two eight-variable tables
mix.array.list = vector(length=2, mode="list")
# first eight variables
# make marginal data frame
first.eight.df =
compact.data.frame(usually.resident.15plus.df[,1:9])

# create list of 3-dimensional arrays, set elements to NULL
array.list = vector(length=56, mode="list")

# now loop, calculating 3-d marginal tables
index = 1
for(i in 2:7){
```

```
  for(j in (i+1):8){
    for(k in (j+1):9){
      array.list[[index]] =
      df2table.mix(first.eight.df[,c(1,i,j,k)])
      index = index + 1
      }
    }
  }
# fit 3-factor interaction model by ipf (816,480 cells)

mix.array.list[[1]] = ipf.fitter(array.list)


# Now do the same thing for the last eight variables

last.eight.df = compact.data.frame(usually.resident.15plus.df[,c(1,4:11)])

# create list, set elements to NULL
array.list = vector(length=56, mode="list")
# now loop
index = 1
for(i in 2:7){
  for(j in (i+1):8){
    for(k in (j+1):9){
      array.list[[index]] = df2table.mix(last.eight.df[,c(1,i,j,k)])
      index = index + 1
      }
    }
  }
# fit 3-factor interaction model by ipf (1,166,400 cells)

mix.array.list[[2]] = ipf.fitter(array.list)


# Fit mixture model

params = mixture.fitter.fusion(mix.array.list, T=10, MAXITER = 100)
```

The current version of this function is written entirely in R and requires a fair bit of time to converge for a problem of this size (it took about 24 hours on my machine). Recoding in a lower-level language would of course improve the performance.

As a check, we calculated the marginal distributions of each variable from the CURF and also from the fitted mixture model. The results are shown in Figure 5. The agreement is excellent.

How should we split the variables into two groups? In the present case, we made a choice based on the alphabetical ordering of the variable names, resulting in the non-overlapping pairs (“AgeGroup”, “BirthPlace”) and (“WorkLabForceStatus”, “YearsAtRes”). However, if we have prior knowledge of the association between the variables, we can use this to make a more informed choice. The conditional association between the non-overlapping pairs, given the other variables, is determined completely by the mixture assumption, as there are no direct observations on the same individuals of these four variables simultaneously. If we suspect that another choice of pairs are conditionally independent, using these would make us less reliant on the mixture assumption.

Finally, to generate 100,000 records from the joint distribution of these 10 variables, and store them in a csv file `usually.resident.15+.csv`, we can use the code

```
N=100000
file = "usually.resident.15+.csv"
generate.sample.mix(N, file, params)
```

4.3.3 The employed usually resident population ages 15 and over

The data frame for this subpopulation has 16 variables (including the counts) and a probability table with 2,821,754,880,000 cells. Fitting a model to this table is a rather daunting task, but a possible approach is the following: We split the 16 variables up into five groups, fit a three-factor interaction model to each group, and then fuse the results together. Consider the following five groups:

Group 1 Variables “AgeGroup”, “BirthPlace”, “EmploymentStatus”, ‘HighestQual”, “IndustryGroup”, “LegalMaritalStatus”, “NumLanguagesSpoken”, with a probability table of 756,000 cells;

Group 2 Variables “EmploymentStatus”, “HighestQual”, “IndustryGroup”, “LegalMaritalStatus”,“NumLanguagesSpoken”, “OccupationCode”, “SectorOfOwnership”, with a probability table of 540,000 cells;

Group 3 Variables “IndustryGroup”, “LegalMaritalStatus”,“NumLanguagesSpoken”, “OccupationCode”, “SectorOfOwnership”, “Sex”, “SocialMaritalStatus”, “TotalHrsWrkd”, with a probability table of 432,000 cells;

Group 4 Variables “NumLanguagesSpoken”, “OccupationCode”, “SectorOfOwnership”, “Sex”, “SocialMaritalStatus”, “TotalHrsWrkd”, “TotalIncomeGroup”, “TravelToWorkGroup”, with a probability table of 777,600 cells.

Group 5 Variables “SectorOfOwnership”, “Sex”, “SocialMaritalStatus”, “TotalHrsWrkd”, “TotalIncomeGroup”, “TravelToWorkGroup”, “WorkLabForceStatus”, “YearsAtRes”, with a probability table of 373,248 cells.

We can repeat the analysis of the last section and fit three-factor interaction models to the CURF data as was done in the case of the usually resident population 15 and over. The code is

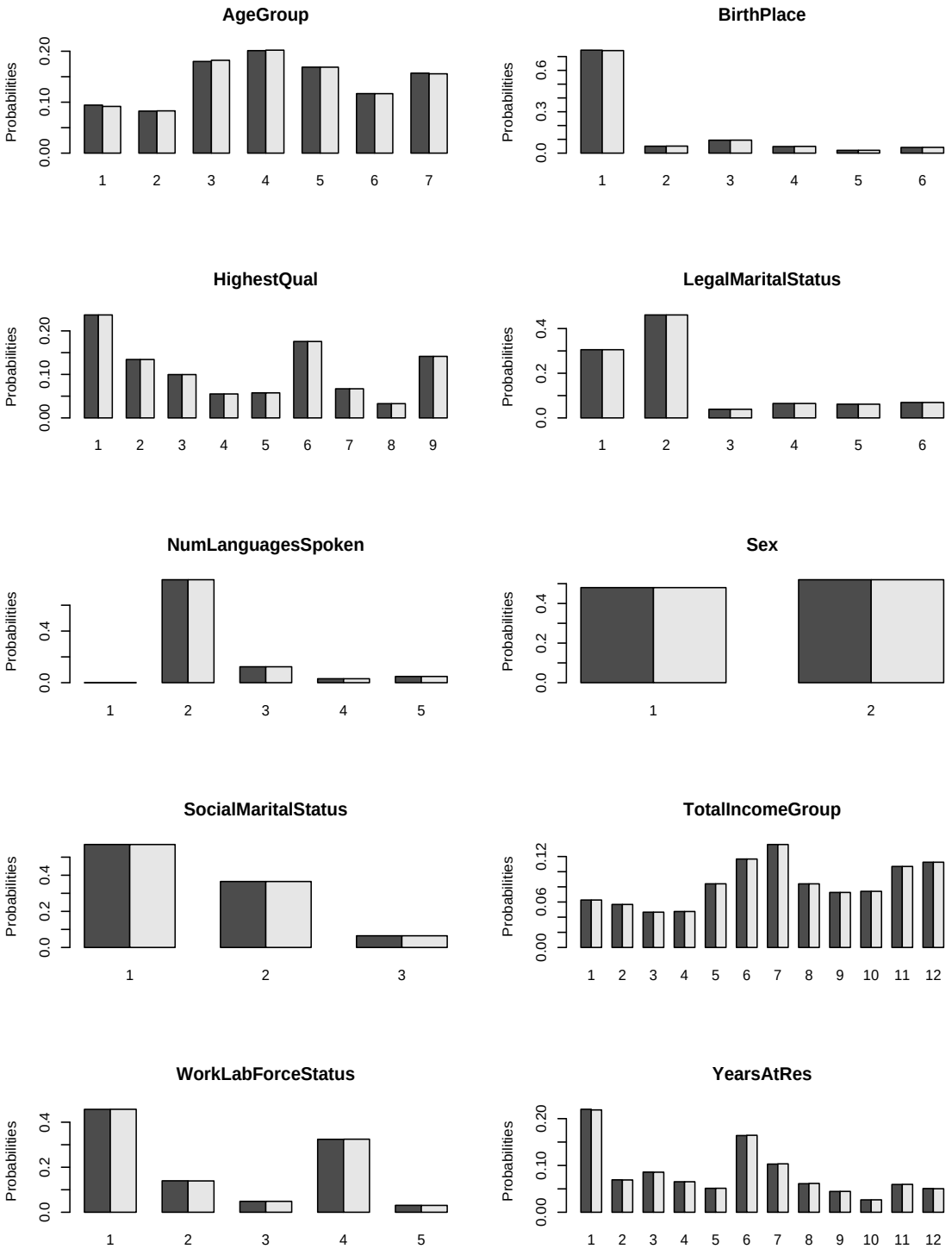

Figure 5: Empirical and fitted marginal probabilities. Grey bars are empirical, black are fitted.

```
# set up list of arrays  (one element per group)for input  into
# function mixture.fitter.fusion

mix.array.list = vector(length=4, mode="list")

# group1
# group 1 variables are in columns 2-8
temp.df = compact.data.frame(
              employed.usually.resident.15plus.df[,1:8])

# create list, set elements to NULL
array.list = vector(length=35, mode="list")
# now loop
index = 1
for(i in 2:6){
  for(j in (i+1):7){
    for(k in (j+1):8){
      array.list[[index]] = df2table.mix(temp.df[,c(1,i,j,k)])
      index = index + 1
      }
    }
  }
mix.array.list[[1]] = ipf.fitter(array.list)

# group2
# group 2 variables are in columns 4-10
temp.df = compact.data.frame(
              employed.usually.resident.15plus.df[,c(1, 4:10)])

# create list, set elements to NULL
array.list = vector(length=56, mode="list")
# now loop
index = 1
for(i in 2:6){
  for(j in (i+1):7){
    for(k in (j+1):8){
      array.list[[index]] = df2table.mix(temp.df[,c(1,i,j,k)])
      index = index + 1
      }
    }
  }
mix.array.list[[2]] = ipf.fitter(array.list)

# group3
# group 3 variables are in columns 6-13
```

```
temp.df = compact.data.frame(
              employed.usually.resident.15plus.df[,c(1, 6:13)])

# create list, set elements to NULL
array.list = vector(length=56, mode="list")
# now loop
index = 1
for(i in 2:7){
  for(j in (i+1):8){
    for(k in (j+1):9){
      array.list[[index]] = df2table.mix(temp.df[,c(1,i,j,k)])
      index = index + 1
      }
    }
  }
mix.array.list[[3]] = ipf.fitter(array.list)

# group4
# group 4 variables are in columns 8-15
temp.df = compact.data.frame(
              employed.usually.resident.15plus.df[,c(1, 8:15)])

# create list, set elements to NULL
array.list = vector(length=56, mode="list")
# now loop
index = 1
for(i in 2:7){
  for(j in (i+1):8){
    for(k in (j+1):9){
      array.list[[index]] = df2table.mix(temp.df[,c(1,i,j,k)])
      index = index + 1
      }
    }
  }
mix.array.list[[4]] = ipf.fitter(array.list)

# group5
# group 5 variables are in columns 10-17
temp.df = compact.data.frame(
              employed.usually.resident.15plus.df[,c(1, 10:17)])

# create list, set elements to NULL
array.list = vector(length=56, mode="list")
# now loop
index = 1
```

```
for(i in 2:7){
  for(j in (i+1):8){
    for(k in (j+1):9){
      array.list[[index]] = df2table.mix(temp.df[,c(1,i,j,k)])
      index = index + 1
      }
    }
  }
mix.array.list[[5]] = ipf.fitter(array.list)



# now fit the mixture model

params = mixture.fitter.fusion(mix.array.list, T=10, MAXITER = 100)
```

When fitting the five group tables using IPF, we have just used the CURF margins for simplicity. The Tablebuilder margins could be substituted where available, as in the case of the usually resident population example. However, even just using the CURF data, the agreement with the empirical data is very good, as can be seen from the marginal distributions of the empirical and fitted probabilities shown in Figure 6.

Finally, to generate say 100,000 records from this distribution, and store them in a file `employed.usually.resident.15+.csv` we type

```
N=100000
file = "employed.usually.resident.15+.csv"
generate.sample.mix(N, file, params)
```

Our final illustration also involves the employed usually resident population aged 15 and over. Statistics New Zealand produces a series of synthetic data sets known as SURF's for use in schools. There is one based on the 2006 census, having 11 variables, plus area information. There are 300 records for each of 16 districts, making 4800 records in all.

Of these variables, seven occur on our list. To fit a model to all 11 variables, a combination of the the IPF or mixture approach could be used, using actual census data to create the input tables. The code required is similar to that in the examples above. For a more concrete illustration, consider the problem of fitting a model to the seven SURF variables that are on our list. The variables are "AgeGroup", "HighestQual", "Sex", "TotalHrsWrkd", "TotalIncomeGroup", "TravelToWorkGroup", "WorkLabForceStatus", and the complete table has 373,248 cells, well within the range of the IPF method. One approach is to fit a log-linear model having all interactions up to a certain order, using the CURF data (or any available microdata) to calculate the marginal tables. Which model should we fit? Here, there is a trade-off between high order (which will give a richer approximating set) and low order (the lower-order margins can be estimated with less

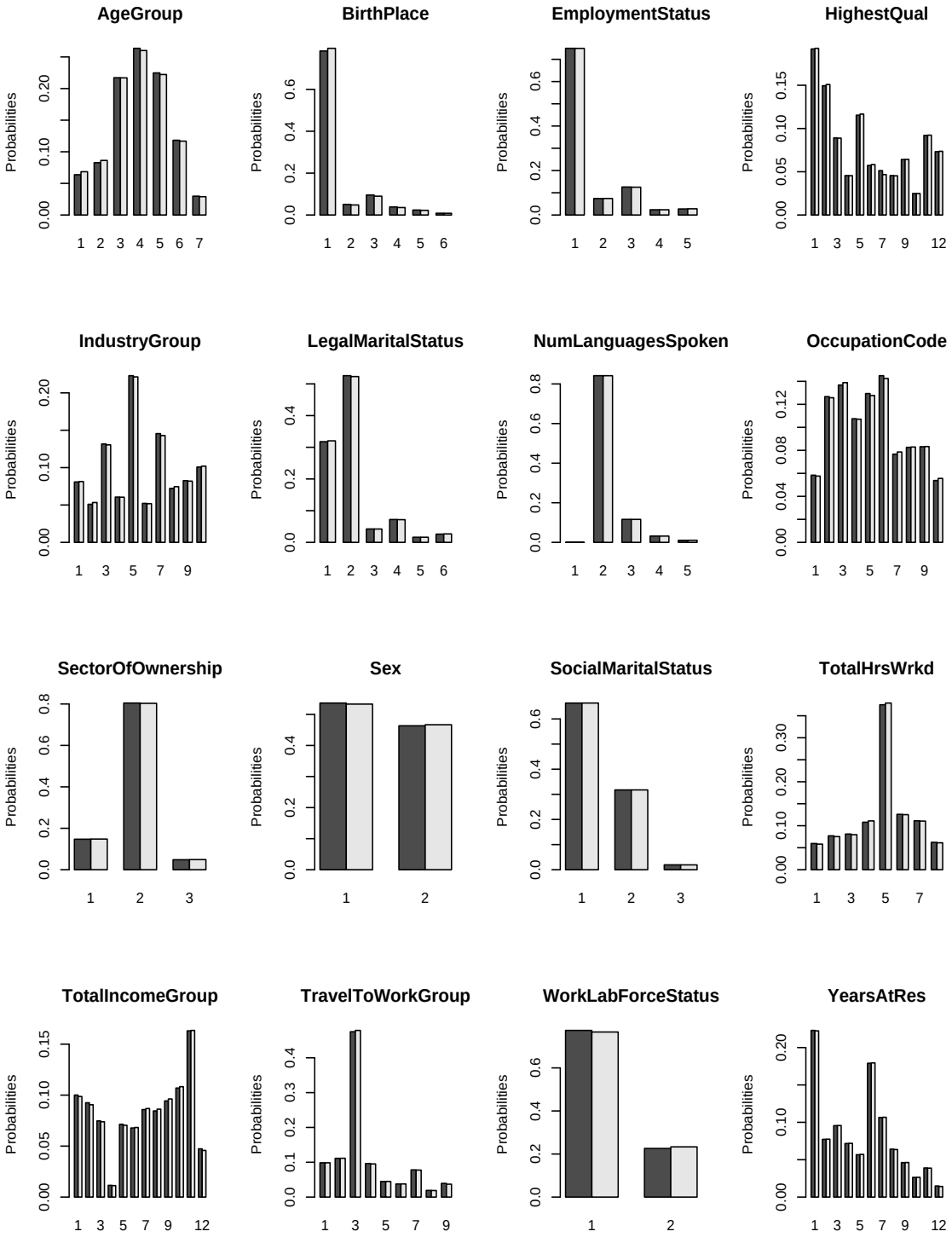

Figure 6: Empirical and fitted marginal probabilities for the employed usually resident population aged 15 and over. Grey bars are empirical, black are fitted.

error from the CURF.) We can calculate the AIC for models each having different orders and choose the model with smallest AIC. From Table 10, we see that a two-factor model is indicated by the AIC criterion, although this may be different with a bigger set of microdata. The model is fitted and the data generated as before:

Table 10: AIC for different models.

Order	AIC	No.of parameters
1	372874.4	48
2	331254.6	942
3	342666.3	9696
4	421063.6	55857
5	662561.0	179472
6	941436.5	319040

```
# fit  models of order 2
#  loop over pairs
index = 1
array.list = vector(length=21, mode="list")
for(i in 2:7){
  for(j in (i+1):8){
    array.list[[index]] = df2table.mix(surf.df[,c(1,i,j)])
    index = index + 1
    }
  }
prob.table2 = ipf.fitter(array.list)
N = 4800
file ="surf.csv"
generate.sample.prob(N, file, prob.table2)
```

The AIC is calculated using the function `param.count`, which calculates the number of parameters in a log-linear model having interactions up to a certain order (i.e order=2 corresponds to having all two-way interactions, but none higher.)

```
no.of.params = param.count(Kvec, order=2)
AIC2  = 2*(no.of.params - dim(surf.df)[1]*sum(prob.table*log(prob.table2)))
> AIC2
[1] 331254.6
```

.

5 SUMMARY AND CONCLUSIONS

In this report, we have discussed a variety of methods that can be used to generate synthetic data of quite high dimension. The methods rely on fitting standard statistical models to tables of relative frequencies, and generating data from these models. In particular, we use log-linear models (fitted by either IPF or Fisher scoring, and able to handle tables of up to six million cells) and mixture models (fitted by the EM algorithm, also handling tables of up to six million cells).In addition, we discussed how to fit mixture models to sets of marginal relative frequencies, which enables much larger tables to be fitted. We illustrated the application of this method to a table having 48 million cells and an even bigger table having approximately 2.8×10^{12} cells and 16 variables.

The methods have been implemented in R, and a set of functions have been written that are a reasonable compromise between memory use and speed. We provided two functions for fitting log-linear models, one using IPF that requires that the complete probability table be stored in the R workspace, and another based on Fisher scoring that is limited by the number of model parameters but not the number of cells. Two functions for fitting mixtures were also provided; one that fits a mixture to single complete probability table (which is stored in the workspace) and another for fitting a mixture to a set of marginal tables. In the latter case the input tables must fit in the R workspace, but not the fitted table. This function is presently coded completely in R, and runs rather slowly for really big problems. Recoding in C or FORTRAN will be a future project. We have also provided a set of functions for generating data from these fitted models, using a variety of methods.

Finally, we have supplied a set of files derived form the 2001 Census of Population and Dwellings, that can be used as suitable input files for the illustration of our methods. Overall, the methods do seem practical for the generation of synthetic data sets of moderate size whose characteristics match those of the real population with reasonable accuracy.

These methods can certainly be improved. One obvious candidate for improvement is program execution speed. This currently is reasonable for the IPF models, but needs improvement in the case of fitting mixture models. As noted above, recoding the mixture functions in a lower-level language is an obvious remedy.

Further work needs to be done in assessing the goodness of fit of models to high dimensional tables which are necessarily sparse. This is a difficult but important problem. Some success here would permit a better determination of how well the models we describe fit the actual census microdata, and how valid inference based on the resulting synthetic data sets will be.

Finally, the hybrid methods we have used rely on the division of variables into groups. Further work needs to be done on how the choice of groups impacts the goodness of fit, and how well all or selected higher dimensional margins are preserved.

REFERENCES

Agresti, A. (2002). *Categorical Data Analysis, 2nd Ed.* Johns Hopkins University Press, Baltimore.

Bishop, Y.M.M., Fienberg, S.E. and Holland, P.W. (1975). *Discrete Multivariate Analysis*. MIT Press, Cambridge.

Christensen, R. (1997). *Log-Linear Models and Logistic Regression*. Springer-Verlag, New York.

Deming, W.E and Stephan, F.F. (1940). On a least squares adjustment of a sampled frequency table when the expected marginal totals are known. *Annals of Mathematical Statistics*, **11**, 427 – 444.

Graham, P. and Penny, R. (2007). Multiply imputed synthetic data files. *Official Statistics Research Series, Vol 1.*

Graham, P. and Penny, R. (2008). Methods for creating synthetic data. *Official Statistics Research Series, Vol 3.*

Hastings, W.K. (1970). Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, **57**, 97 – 109.

Jackson, L.F. (2007). Uniques and disclosure control design. *Official Statistics Research Series, Vol 1.*

Jackson, L.F. and Gray, A. (2007). Impact of global recoding to preserve confidentiality on information loss and statistical validity of subsequent data analysis. *Official Statistics Research Series, Vol 2.*

Kamakura, W. A. and Wedel, M. (1997). Statistical data fusion for cross-tabulation. *Journal of Marketing Research*, **34**, 485–498.

Lee, A. J. (2007). Generating synthetic unit-record data from published marginal tables. *Official Statistics Research Series, Vol 1.*

R Development Core Team (2008). *R: A language and environment for statistical computing.* R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org.

Reiter, J. P. (2005). Using CART to generate partially synthetic, public use microdata. *Journal of Official Statistics* **21**, 365–377.

Ripley, B. D. (1987). *Stochastic Simulation*. Wiley, New York.

Woodcock, S. D. and Benedetto, G. (2007). Distribution-preserving statistical disclosure limitation. Unpublished MS.

A Appendices

A.1 Description of the R functions

In this appendix we document the following R functions:

```
as.standard
calculate.probs
check.data.frame
compact.data.frame
df2table
df2table.mix
generate.sample.mix
generate.sample.MH
generate.sample.prob
is.standard
ipf.fitter
KLDist
log.lin2table
log.lin.fitter
make.all
make.margin
mixture.fitter.single
mixture.fitter.fusion
params2table
param.count
table2df
```

as.standard	package:SNZ	R documentation

Coerces a data frame into standard form

Description:

Coerces a data frame representing a contingency table into a standard form.

Usage:

```
as.standard(data.df)
```

Arguments:

data.df: a data frame.

Details:

If the data frame represents a contingency table (i.e. if one variable contains counts and the rest are factors) it is transformed into standard form. Otherwise an error message is returned.

Value:

A data frame in standard form.

Example:

```
## convert data frame data.df to standard form
data.df = as.standard(data.df)
```

calculate.probs	package:SNZ	R documentation

Calculates a table of fitted probabilities from a log-linear parameter vector

Description:

Takes the log-linear parameter vector and calculates the table of fitted probabilities.

Usage:

```
calculate.probs(beta, Kvec)
```

Arguments:

beta: The parameter vector produced by the function log.lin.fitter.

Kvec: A vector giving the factor lengths.

Details:

The parameter vector beta is the result of fitting a two-factor interaction model, assuming the factors are in the order implied by the vector Kvec.

Value:

An array of dimension Kvec containing a table of probabilities.

Example:

check.data.frame	package:SNZ	R documentation

Checks a data frame

Description:

Checks if a data frame is in standard form.

Usage:

```
check.data.frame(data.df)
```

Arguments:

data.df: a data frame.

Value:

A list with elements

status: has value 1 if the argument can be coerced into a data frame representing a contingency table in standard form, and zero otherwise;

data: if status = 1, the data frame representing the contingency table, otherwise NULL;

message: if status = 0, an error message.

Example:

```
## check data frame data.df to see if it represents
## a contingency table in standard form
data.df=check.data.frame(data.df)$data
```

compact.data.frame	package:SNZ	R documentation

Combine rows of a data frame

Description:

Combines rows of a data frame with counts in the first column. Rows that are identical except for the counts are combined by adding the counts together.

Usage:

```
combine.data.frame(data.df)
```

Arguments:

data.df: a data frame in standard form, except may contain rows that are identical except for the counts.

Value:

A data frame in standard form.

Details:

The input data frame will usually be a data frame from which some of the columns (not the counts) have been deleted. The function is useful for creating data frames in standard form representing marginal tables.

Example:

```
 # standard.df is a data frame representing # a 4-dimensional table
# Now make a df in standard form containing the
# marginal table of the first two variables
margin.df = compact.data.frame(standard.df[,c(1,2,3)])
```

df2table	package:SNZ	R documentation

Converts a data frame in standard form into an array

Description:

Converts a data frame in standard form into an array, and issues an error message if the input cannot be converted to standard form.

Usage:

```
df2table(data.df)
```

Arguments:

data.df: a data frame.

Details:

If the data frame represents a contingency table (i.e. if one variable contains counts and the rest are factors) it is transformed into an array. Otherwise an error message is returned.

Value:

An array representing the contingency table.

Example:

```
## convert data frame data.df to an array my.table
my.table = df2table(data.df)
```

df2table.mix	package:SNZ	R documentation

Converts a data frame in standard form into an array, mixing it with an independence table constructed from the margins of the array

Description:

First converts a data frame in standard form into an array, and issues an error message if the input cannot be converted to standard form. Then it calculates the one-dimensional margins of the array, forms an independence table, and then forms the mixture (with proportions τ and $(1-\tau)$) of the initial table and the independence table.

Usage:

```
df2table.mix(data.df, tau=0.99)
```

Arguments:

`data.df`: a data frame.

`tau`: the mixing probability, a number between 0 and 1. Default is 0.99.

Details:

The initial and independence tables are stored in the R workspace so that this function cannot be used with very large tables.

Value:

An array representing the table of probabilities of the resulting mixture distribution.

Example:

```
## convert data frame data.df to an array my.table, mixing it with
an independence table
my.table = df2table.mix(data.df, tau=0.99)
```

generate.sample.MH	package:SNZ	R documentation

Generates a data frame by sampling from a log-linear model

Description:

Creates a file of synthetic data of specified size by sampling with replacement from a the probability distribution implied by a two-factor interaction log-linear model.

Usage:

```
generate.sample.MH(N, file, beta, margin.list, thin=10, burn = 1000)
```

Arguments:

N: The number of records to be generated.

file: The name of the file where the data are to be stored.

beta: The parameter vector, usually calculated by the function log-lin.fitter.

margin.list: A list containing the one-dimensional marginal distributions of the log-linear model, calculated by the function make.margin.

thin: The thinning period for the Metropolis-Hastings algorithm. e.g. if thin =10, retain every 10th value form the Markov chain.

burn: The burn-in period for the Metropolis-Hastings algorithm.

Details:

The function produces a csv file of N records. Each record consists of the values $(i_1, \ldots, i_K)$ of the variables corresponding to the elements of the list margin.list , and is the result of running the Markov chain, retaining the values as specified by thin. Each state of the chain corresponds to a cell of the probability table implied by the parameter vector beta.

Value:

The function returns no value but a csv file containing N records is created.

Example:

```
 Kvec<-c(3,4,2)
N=50000
file = "mydata.csv"
y = runif(prod(array.dim)))
y=y/sum(y)
write.file(y,"probfile")
beta = log.lin.fitter("probfile", Kvec)
margin.list=make.margin("probfile", Kvec)
generate.sample.MH(N, file, beta, margin.list)
```

generate.sample.mix	package:SNZ	R documentation

Generates a data frame by sampling from a mixture model

Description:

Creates a file of synthetic data of specified size by sampling with replacement from a the probability distribution implied by a mixture model.

Usage:

```
generate.sample.mix(N, file, params)
```

Arguments:

N: The number of records to be generated.

file: The name of the file where the data is to be stored.

params: The parameters of the mixture model, as calculated by the functions mixture.fitter.single or mixture.fitter.fusion.

Details:

The function produces a csv file of N records. Each record consists of the values $(i_1, \ldots, i_K)$ of the variables corresponding to the elements of the list margin.list , and is the result of sampling with replacement from the mixture distribution implied by the parameters in params.

Value:

The function returns no value but a csv file containing N records is created.

Example:

```
 Kvec<-c(3,4,2)
N=50000
T=2
file = "mydata.csv"
y = runif(prod(array.dim)))
y=y/sum(y)
prob.table<-array( y, dim=Kvec, dimnames =
list(A = paste("a",1:Kvec[1], sep=""),
B=paste("b",1:Kvec[2], sep=""), C=paste("c",1:Kvec[3], sep="")))
params= mixture.fitter.single(prob.table, T)
generate.sample.mix(N, file, params)
```

generate.sample.prob	package:SNZ	R documentation

Generates a data frame by sampling from a probability table

Description:

Creates a file of synthetic data of specified size by sampling with replacement from a specified probability table.

Usage:

```
generate.sample.prob(N, file, prob.table)
```

Arguments:

N : The number of records to be generated.

file: The name of the file where the data is to be stored.

prob.table: The probability table.

Details:

The function produces a csv file of N records. Each record consists of the values $(i_1, \ldots, i_K)$ of the variables corresponding to the dimensions of the array containing the probability table, and is the result of selecting a single cell i with the probabilities in prob.table.

Value:

The function returns no value but a csv file containing N records is created.

Example:

```
 Kvec<-c(3,4,2)
N=50000 file = "mydata.csv" y = runif(prod(array.dim)))
prob.table<-array( y, dim=Kvec, dimnames =
list(A = paste("a",1:Kvec[1], sep=""),
B=paste("b",1:Kvec[2], sep=""), C=paste("c",1:Kvec[3], sep="")))

generate.sample.prob(N, file, prob.table)
```

is.standard	package:SNZ	R documentation

Tests if argument is a data frame in standard form

Description:

Checks to see if the argument is a data frame in standard form, and returns TRUE or FALSE accordingly.

Usage:

```
is.standard(data.df)
```

Arguments:

data.df: The object to be tested

Details:

The function tests if the argument is a data frame, checks that the first column contains counts, and that the remaining columns are all factors.

Value:

A logical value

Example:

```
is.standard(data.df)
```

ipf.fitter	package:SNZ	R documentation

Fits a log-linear model defined by a set of margins using IPF

Description:

Fits the a log-linear model defined by a set of margins, which are in the form of a list of relative frequency tables, using IPF

Usage:

```
ipf.fitter(array.list, MAXITER = 20)
```

Arguments:

`array.list`: A list of arrays, where each array contains a table of relative frequencies summing to one.

`MAXITER`: The number of IPF iterations to be performed.

Details:

The function performs `MAXITER` iterations of the IPF algorithm, adjusting each margin in the list in turn. The arrays in `array.list` must have compatible variable names and factor levels.

Value:

An array containing the fitted probability table.

Example:

```
 # margins are in data frames income_occupation_sex.df,
# income_hours_sex.df, hours_occupation_sex.df,
# hours_income_occupation.df
# convert dfs to arrays of relative frequencies

temp = df2table(income_occupation_sex.df)
income_occupation_sex = temp/sum(temp)

temp = df2table(income_hours_sex.df)
income_hours_sex = temp/sum(temp)

temp = df2table(hours_occupation_sex.df)
hours_occupation_sex = temp/sum(temp)

temp = df2table(hours_income_occupation.df)
hours_income_occupation = temp/sum(temp)

# make the list
array.list = list(hours_occupation_sex, hours_income_occupation,
income_occupation_sex, income_hours_sex)

# calculate the fitted table of relative frequencies:
result.table = ipf.fitter(array.list, MAXITER=100)
```

KLDist	package:SNZ	R documentation

Computes Kullback-Leibler divergence

Description:

Calculates the Kullback-Leibler divergence between the empirical frequencies in `freq.array` and the mixture model specified by `params`.

Usage:

```
KLDist(freq.array, params)
```

Arguments:

`freq.array`: an array of relative frequencies.

`params`: data structure returned by `mixture.fitter single` or `mixture.fitter.fusion` containing the parameters of a mixture model

Details:

The function expects that the orders of the factors and levels in each argument are identical.

Value:

The KL divergence between the two distributions.

Example:

```
params = mixture.fitter.single(freq.array, T=10)
KLDist(freq.array,params)
```

loglin2table	package:SNZ	R documentation

Calculates fitted probability table from the parameters of a log-linear model

Description:

Calculates the fitted probability table when supplied the parameter vector, the numbers of levels for each factor and optionally a list of factor levels.

Usage:

```
loglin2table(beta, Kvec, dimnames=NULL)
```

Arguments:

beta: The parameter vector, usually created by the function log.lin.fitter.

Kvec: A vector giving the dimensions of the array representing the probability table, or, equivalently, the number of levels for each of the factors in the corresponding standard data frame.

dimnames: The dimnames attribute of the array, a list of factor levels.

Details:

The function produces an array representing the fitted probability table corresponding to the parameter vector beta. Care must be taken to ensure that the array dimensions in Kvec and the factor levels in dimnames are compatible with the ordering of the parameters in beta.

Value:

An array representing the probability table.

Example:

```
 # suppose beta is the parameter vector in example 9
#the following calculates the corresponding fitted probability table
Kvec = c(2,3,4,3,6)
dimnames = list(A=1:Kvec[1], B=1:Kvec[2], C=1:Kvec[3], D=1:Kvec[4],
E=1:Kvec[5])
result.table = loglin2table(beta, Kvec, dimnames)
```

log.lin.fitter	package:SNZ	R documentation

Fits a log-linear model by the Fisher scoring method

Description:

Fits a log-linear model by the Fisher scoring method, to a set of counts contained in a file

Usage:

```
log.lin.fitter(countfile, Kvec, MAXITER=20, TOL=1.0e-6)
```

Arguments:

countfile: A text file containing the complete set (including zeroes) of counts or relative frequencies in reverse lexicographic order as determined by Kvec.

Kvec: A vector giving the dimensions of the array representing the probability table, or, equivalently, the number of levels for each of the factors in the corresponding standard data frame.

MAXITER: The maximum number of Fisher scoring iterations.

TOL: The convergence criterion: if successive iterations of beta differ by less than TOL, the iterations terminate.

Details:

The function fits a log-linear model with all 3- and higher-order interactions zero, using Fisher scoring. The counts, assumed to be in reverse lexicographic order, are read from the file countfile, and the model matrix is generated line by line and never stored. The score vector and information matrix are recalculated at each iteration, reading in the data from the file each time. Thus, the complete set of counts never needs to be stored in the R workspace, so the limiting factor as far as memory is concerned is the number of parameters, not the number of cells. The function returns the parameter vector of the fitted model, with the oder of the parameters consistent with the entries in Kvec. The product of the elements of Kvec should equal the number of entries in the file countfile. The counts are read in a line at a time and the score vector and information matrix accumulated, with each line being processed as a block.

Value:

The parameter vector beta of the fitted model.

Example:

```
 test.df = data.frame(Count=rpois(prod(Kvec),10),
expand.grid(A=factor(1:2),B=factor(1:3),
C=factor(1:4),D=factor(1:3),E=factor(1:6)))
countfile = "testfile.txt"
write(test.df$Count, countfile, ncolumns=100)
Kvec=c(2,3,4,3,6)
beta = log.lin.fitter(countfile, Kvec)
```

make.all	package:SNZ	R documentation

Creates a set of data frames containing the Tablebuilder tables

Description:

Creates a set of data frames containing the Tablebuilder tables in the R workspace

Usage:

```
make.all(path)
```

Arguments:

path: The path of the supplied directory SNZtables. If absent, SNZtables must be the current directory.

Details:

Executing this function causes the set of Tablebuilder tables to be created in the workspace. Note that the path, if given, should use the R double backslash notation.

Value:

None

Example:

```
make.all("F:\\Stats NZ\\2007 project\\SNZTables")
```

make.margin	package:SNZ	R documentation

Calculates the one-dimensional margins of a table of counts stored as a text file

Description:

Creates a list of one-dimensional margins of a table of counts or relative frequencies stored as a text file in reverse lexicographic order, as for the function log.lin.fitter

Usage:

```
make.margin(countfile, Kvec)
```

Arguments:

countfile: A text file containing the complete set (including zeroes) of counts or relative frequencies in reverse lexicographic order as determined by Kvec.

Kvec: A vector giving the dimensions of the array representing the probability table, or, equivalently, the number of levels for each of the factors in the corresponding standard data frame.

Details:

The function produces a list of vectors, one per margin, each vector containing the marginal counts for that factor. Used in conjunction with the function generate.sample.MH.

Value:

A list of vectors, one for each element of Kvec.

Example:

```
 Kvec<-c(3,4,2)
margin.list=make.margin("countfile", Kvec)
```

mixture.fitter.single	package:SNZ	R documentation

Fits a mixture model to single sample

Description:

Fits a mixture model, given a sample from a population in the form of an array of relative frequencies

Usage:

```
mixture.fitter.single(freq.array, T, params0=NULL,
       MAXITER=100, TOL=1.0e-8, trace=TRUE)
```

Arguments:

freq.array An array containing the sample in the form of relative frequencies.

T: The number of mixture components

params0: Initial values for the parameters. If NULL, starting values are generated randomly.

MAXITER: Maximum number of iterations.

TOL: Tolerance, iterations stop of successive values of τ differ by less than TOL.

trace: If true, prints out the iteration number and the change in the vector tau at each iteration.

Details:

The function fits a mixture model with T components by maximum likelihood, using an iteration scheme equivalent to the EM algorithm.

Value:

A list containing elements

tau: The mixing probabilities.

theta: A list of $I_k \times T$ matrices. The tth column of the kth matrix contains the distribution of A_k for the tth component.

dimnames: A named list containing the factor levels.

Example:

```
 A = df2table(income_hours_sex.df)
A = A/sum(A) # convert to relative frequencies
params=mixture.fitter.fusion(A, T=10, params0 = NULL, MAXITER=100)
```

mixture.fitter.fusion	package:SNZ	R documentation

Fits a mixture model to a set of overlapping samples

Description:

Fits a mixture model, given a set of samples from a population with different but overlapping sets of variables.

Usage:

```
mixture.fitter.fusion(array.list, T, params0=NULL, MAXITER=100, TOL=1.0e-8,
trace=TRUE)
```

Arguments:

array.list A list, each member of which is an array containing the sample relative frequencies.

T: The number of mixture components

params0: Initial values for the parameters. If NULL, starting values are generated randomly.

MAXITER: Maximum number of iterations.

TOL: Tolerance, iterations stop of successive values of τ differ by less than TOL.

trace: If true, prints out the iteration number and the change in the vector tau at each iteration.

Details:

The function fits a mixture model with T components by maximum likelihood, using an iteration scheme equivalent to the EM algorithm.

Value:

A list containing elements

tau : The mixing probabilities.

theta: A list of $I_k \times T$ matrices. The tth column of the kth matrix contains the distribution of A_k for the tth component.

dimnames: A named list containing the factor levels.

Example:

```
 A = df2table(income_hours_sex.df)
A = A/sum(A) # convert to relative frequencies
B = df2table(income_occupation_sex.df)
B = B/sum(B)# convert to relative frequencies
array.list = list(A,B)
params=mixture.fitter.fusion(array.list, T=10,
         params0=NULL, MAXITER=100)
```

params2table	package:SNZ	R documentation

calculates a probability table from a set of mixture parameters

Description:

Calculates the probability table corresponding to a set of mixture parameters.

Usage:

```
params2table(params)
```

Arguments:

params : A list containing elements `tau`, `theta` and `dimnames` as specified in the output of `mixture.fitter.single` above.

Details:

The function produces an array containing the probability table corresponding to the mixture parameters in `params`, and labels the array with the list `dimnames` if present in the list `params`.

Value:

An array containing the fitted probability table.

Example:

```
 A = df2table(income_hours_sex.df)
A = A/sum(A) # convert to relative frequencies
params=mixture.fitter.fusion(A, T=10,
       params0 = NULL, MAXITER=100) params2table(params)
```

param.count	package:SNZ	R documentation

Counts the number of parameters in a log-linear model

Description:

Counts the number of parameters in a log-linear model, given the order of the model and the number of levels for each factor. (A model has order 2 if there are no 3- and higher-order interactions present, order 3 if there are no 4- and higher-order interactions present, and so on.)

Usage:

```
param.count(Kvec, order=2)
```

Arguments:

Kvec: A vector giving the dimensions of the array representing the probability table, or, equivalently, the number of levels for each of the factors in the corresponding standard data frame..

order: the oder of the log-linear model.

Details:

The number of parameters (including the constant term) of the log-linear model of given order is calculated, using a simple combinatorial formula.

Example:

```
## calculate the number of parameters in a 2-factor interaction model
param.count(Kvec)
```

table2df	package:SNZ	R documentation

Converts an array to a data frame in standard form

Description:

Converts an array into a data frame in standard form, and issues an error message if the input is not capable of conversion to standard form (e.g. if there are negative or non-integer entries in the array.)

Usage:

```
table2df(my.table)
```

Arguments:

my.table: a multidimensional array.

Details:

If the array represents a contingency table (i.e. if the entries are non-negative integers) it is transformed into a data frame in standard form. Otherwise an error message is issued. The count variable in the data frame is named y.

Example:

```
## convert array my.table to a data frame data.df in standard form
data.df = table2df(my.table)
```

A.2 Installation of the R functions

We have provided a file of the R functions used in the report, in the file `SNZ2008.r`. These include the functions documented above, and also some other utility functions called by these. The functions should be be run under Windows.

To use these functions, simply unzip the file `SNZ2008.zip`, place the contents (including the R source code in file `SNZ2008.r`) in some suitable folder. The zip file contains the file `SNZ2008.r` along with two folders, `SNZTables` and `CURF`. The first of these folders contains R scripts and files for producing R versions of the Tablebuilder files as described in Section 4.2.1, while the second contains R scripts and files for producing data frames containing subsets of the CURF variables. Their use is described in Section 4.2.2.

To use the functions, invoke R, choose “Source R code...” from the R file menu, and select the file `SNZ2008.r`. You are then ready to go. You can follow the discussion in Sections 4.2.1 and 4.2.2 to read in the example data from the Tablebuilder files and the CURF, in the form of R data frames.

A.3 R function listings

In this appendix, we provide a listing of the R functions in the file `SNZ2008.r`.

```
# Functions for Windows version, 2008 project

#############################################################################
#
check.data.frame=function(data.df){

# checks that the data frame is in standard form
# returns a list with elements status
# (=0 if arg cannot be coereced into the standard form, otherwise 1)
# and data ( the possibly modified data frame in standard form,
# if status =1, otherwise NULL)
# standard form is a data frame with the first column a numeric variable,
# and the other variables factors.

# The numeric variable must be a non-negative vector of counts.

# first check argument is a data frame, quit if not

if(!is.data.frame(data.df))return (list(status=0, data=NULL,
 message = "Not a data frame"))

n.var<-dim(data.df)[2]
is.number=logical(n.var)
for(j in 1:n.var) is.number[j]<-is.numeric(data.df[,j])

# check if exactly one is numeric
if(!(sum(is.number)==1)) return (list(status=0, data=NULL, message =
"More than one numeric variable"))

# check to see if y is non-neg
if(!all(data.df[,is.number]>=0)) return (list(status=0, data=NULL,
message ="Cannot have non-negative counts"))

# check that all factors have more than one level
use.cols = (1:n.var)[!is.number]
max.levels = numeric(length(use.cols))
for(i in 1:length(use.cols)) max.levels[i]=length(levels(data.df[,i]))
if(any(max.levels==1))stop(paste("All factors must have at least 2 levels.\n
The following variables have only one:",
                          paste(var.names[max.levels==1], collapse=", ")) )

# now re-arrange factors into alphabetic order
# first get name of count variable
```

```
count.name = names(data.df)[is.number]
data.df = data.frame(data.df[,is.number],
data.df[, use.cols[order(names(data.df)[!is.number])]])
names(data.df)[1] = count.name
# and counts in reverse lexographic order

dims = get.dim(data.df)
sortvec = rep(1,dim(data.df)[1]); myprod = 1
for(l in 1:length(dims)){
sortvec=(unclass(data.df[,l+1]) - 1)*myprod + sortvec
myprod = myprod * dims[l]
}

# combine duplicated rows
data.df = data.df[order(sortvec),]
sortvec = sort(sortvec)
no.dups = !duplicated(sortvec)
counts = tapply(data.df[,1], sortvec, sum)
data.df = data.df[no.dups, ]
data.df[,1] = counts

list(status=1, data=data.df, message ="Data frame OK")
}

# gets the number of levels of the factors in the
# standard data frame data.df

get.dim = function(data.df){

J = dim(data.df)[2] -1  # assumes counts are in column 1
dims=numeric(J)
for(j in 1:J)dims[j] = length(levels(data.df[,j+1]))
dims
}

#################################################################
#
is.standard = function(data.df){

# wrapper for check.data.frame, tests if object is
# a data frame representing a table
check.data.frame(data.df)$status ==1
}

#################################################################
#
as.standard = function(data.df){
```

```
# wrapper for check.data.frame, tests if object is a
# data frame representing a table, returns table in
# standard form if so

result =  check.data.frame(data.df)
if(result$status!=1)stop(result$message)
result$data
}
#################################################################
#
table2df<- function(table, include.zero=FALSE){

# function to convert an array "table" to a data frame
# in standard form

# converts a numerical array to a data frame

# First check input is an array

if(!is.array(table))stop("Input must be an array")

# then check presence of  dimnames

dimlist=dimnames(table)
if(is.null(dimlist)){
n.var<-length(dim(table))
var.names= paste("V", 1:n.var, sep="")
dimlist<-vector(n.var, mode="list")
for(i in 1:n.var) dimlist[[i]] = as.character(1:(dim(table)[i]))
names(dimlist)=var.names
}
y=as.vector(table)
if(any(y<0))stop("Table must have non-negative counts")
data.df=data.frame(y, expand.grid(dimlist))
if(include.zero) data.df else data.df[y!=0,]
}

#####################################################################
#
df2table = function(data.df){

# converts data frame to array
# first checks that the data frame is in standard form,
# if not, coerces into standard form, bails out if not possible.
# then forms array after resoring missing zero counts

# first convert input to standardised data frame, quit if not
```

```
data.df = as.standard(data.df)


# now restore zero counts
dims = get.dim(data.df)
sortvec = rep(1,dim(data.df)[1]); myprod = 1
for(l in 1:length(dims)){
sortvec=(unclass(data.df[,l+1]) - 1)*myprod + sortvec
myprod = myprod * dims[l]
}
counts = numeric(prod(dims))
counts[sortvec] = data.df[,1]

# get levels for each factor

levels.list = vector(mode="list", length = length(dims))
for ( i in 1:length(dims))levels.list[[i]] = levels(data.df[,i+1])
names(levels.list) = names(data.df)[-1]
array(counts, dims, dimnames=levels.list)
}

#####################################################################
#
n.to.nvec = function(n, base.vec){

# converts the integer n into a mixed-base representation
# bases are in base.vec
# not called by users

# eg  if (n-1) = i-1 + (j-1)*I + (k-1)*I*J + (l-1)*I*J*K
# returns (i-1,j-1,k-1,l-1)

Nvar=length(base.vec)
l.index = numeric(Nvar)
cumlevels = (c(1,cumprod(base.vec)[-Nvar]))
nn=n-1
for(index in length(base.vec):1){
l.index[index] = nn%/%cumlevels[index]
nn = nn %% cumlevels[index]
}
l.index
}




#####################################################################
#
```

```
ipf.fitter = function(array.list, MAXITER=20){


# program to calculate cell probabilities using IPF, all R version
# inputs:  a single list of arrays
# MAXITER: maximum number of IPF iterations

if(!is.list(array.list))stop(
      "First argument must be a list of relative frequency tables")




for(i in 1:length(array.list)){
 if(!is.probarray(array.list[[i]])) stop(
"First argument must be a list of relative frequency tables")
}

for(i in 1:length(array.list)){
if(is.null(dimnames(array.list[[i]]))) stop(
"Arrays must have non-null dimension names")
}




if((MAXITER<1)&(round(MAXITER)!=MAXITER))stop(
"MAXITER must be a positive integer")

# get names and dimensions
names.list = vector(mode="list", length=length(array.list))
dim.list = vector(mode="list", length=length(array.list))

vars=NULL; dims = NULL

for(i in 1:length(array.list)) {
names.list[[i]] = names(dimnames(array.list[[i]]))
dim.list[[i]] = sapply(dimnames(array.list[[i]]), length)
vars = c(vars, names.list[[i]])
dims = c(dims, dim.list[[i]])
}

# remove duplicates and sort variable names into order

dup<-duplicated(vars)

var.names<-vars[!dup]
max.levels<-dims[!dup]
```

```
var.order = order(var.names)
var.names = var.names[var.order]
max.levels = max.levels[var.order]

# check for consistency of levels, record factor levels

levels.list = vector(mode="list", length=length(var.names))
for(j in 1:length(var.names)){
  first=TRUE
  name = var.names[j]
  for(i in 1:length(array.list)){
    k = match(name, names(dimnames(array.list[[i]])))
    if( !is.na(k)){
       current.levels = dimnames(array.list[[i]])[[k]]
       if(first) { first.levels = current.levels
   first = FALSE} else
           {
            if(length(first.levels) !=
                length(current.levels)) stop(paste("Factor levels
                for factor", name, "not compatible"))
            if(any(first.levels!= current.levels))
stop(paste("Factor levels for factor", name, "not compatible"))
            }
        }
    }
levels.list[[j]] = first.levels
}


names(levels.list) = var.names
# make a binary matrix representing effect names
# ie 0 1 0 1 represents B:D, 0 1 1 1 B:C:D etc

N.effect = length(array.list)
Nvar = length(var.names)


effect.mat = matrix(FALSE,N.effect, Nvar)
margin = vector(mode="list", length= N.effect)

for (i in 1: N.effect){
effect.mat[i,match(names.list[[i]],var.names)] = TRUE
}

# Now do IPF iterations
x = array(1/prod(max.levels), dim=max.levels)
index = 1:length(max.levels)
```

```
for(iter in 1:MAXITER){
for(i in 1:N.effect){
temp=apply(x, index[effect.mat[i,]],sum)
xfactor = ifelse(temp==0, 0, array.list[[i]]/temp)
x= x*expand(xfactor, max.levels, effect.mat[i,])
}}
dimnames(x) = levels.list
x
}


#################################################################
#
expand = function(x, max.levels, margin){
# expands a marginal table to a complete table
# x: the marginal table
# maxlevels: the vector of factor lengths for the complete table
# margin: a logical vector indicating the factors defining the margin
# ie T,F,F,T indicates the AD margin of the complete ABCD table
# used for IPF, not called directly by users

index = 1:length(max.levels)
temp.levels = c(index[margin],index[!margin])
xx = array(x, dim=max.levels[temp.levels])
perm = match(index, temp.levels)
aperm(xx, perm)
}
#################################################################################
all.poss.combs = function(n,d, chars=c(letters,LETTERS,0-9)){
# calculates all possible combinations of n characters from chars, taken d at a time
# n must be <= 62
if(n>62)stop("Value of n must be less than 62")
sort(if(d==1) paste(chars[1:n], sep="") else
if (n==d) paste(chars[1:d], collapse="") else
c(all.poss.combs(n-1,d,chars), paste(all.poss.combs(n-1,d-1,chars),chars[n], sep="")))
}

#####################################################################
param.count = function(dimvec, order=2){
# calculates number of parameters in model with all interactions of order "order"
chars = c(letters,LETTERS,0-9)
if(!is.dimvec(dimvec))stop("First argment must be a
   vector giving the number of levels for each factor")
if(length(dimvec)>62) stop("too many variables")
if(length(dimvec)<order)stop("length of dimvec<order")

if((order<0)||(order!=round(order))) stop("order must be a positive integer")
```

```
dimvec = dimvec-1
my.sum = 1
for ( i in 1:order){
all.combs = all.poss.combs(length(dimvec), i,chars)
index =  strsplit(all.combs, split="")
index = lapply(index, function(x) match(x, chars))
my.sum = my.sum + sum(unlist(lapply(index, function(aaa) prod(dimvec[as.numeric(unlist(aaa))])
}
my.sum
}




##################################################
nvec.to.n = function(nvec, base.vec){
# converts mixed base representation in nvec into an integer i.e.
# given nvec = i-1, j-1, k-1, l-1 returns

#(n-1) = i-1 + (j-1)*I + (k-1)*I*J + (l-1)*I*J*K

nn = nvec[1]
myprod=1

for(a in 1:(length(base.vec)-1)){
myprod = myprod*base.vec[a]
nn = nn + nvec[a+1]*myprod
}

nn + 1
}


#####################################################################
get.margin.index = function(x,  margin, dimvec){
# represents factor level combination as a mixed-radix number
# margin: a vector of logicals indicating which varables are to be included
# dimvec: vector givoing numbers of factor levels of each variable

  nvec.to.n(x[margin]-1,dimvec[margin])
}

#####################################################################
get.marginal.table = function(y, level.mat, margin, dimvec){

#  computes marginal table as a vector the same dimension as full table
```

```
# y: first column of table.df
# level.mat: table.df as a numerical matrix, first column removed
# margin: a vector of logicals indicating which varables are to be included
# dimvec: vector givoing numbers of factor levels of each variable

index = apply(level.mat, 1, get.margin.index, margin, dimvec)
tapply(y, index, sum)[index]
}




############################################################

make.margin = function(A.df, margin){

# makes marginal table as a data frame in standard form
# margin: vector of indices defining the margin

# check for correct form of A.df


#  unclass factors

for(j in 2:dim(A.df)[2]){
A.df[,j] = if(is.factor(A.df[,j]))unclass(A.df[,j]) else A.df[,j]
}

level.mat = as.matrix(A.df[,-1])
y = A.df[,1]
dimvec = get.dim.num (A.df)

if(length(margin)>length(dimvec))stop(
                    "margin vector wrong length")
if(!is.numeric(margin)) stop(
                    "margin must be vector of indices")

# make hash vector

index=0; prod=1
for( i in 1:length(dimvec[margin])){
   index = index + (level.mat[,margin[i]] - 1)*prod
   prod = prod * dimvec[margin[i]]
}
# compute margins
index = index + 1
counts = tapply(y, index, sum)
```

```
N.margin=length(margin)
dim.margin=dimvec[margin]
level.mat = matrix(0, length(counts), N.margin)
cumlevels = (c(1,cumprod(dim.margin)[-N.margin]))
margin.index=as.numeric(names(counts)) - 1
for(i in length(dim.margin):1){
        level.mat[,i] = margin.index%/%cumlevels[i]
        margin.index = margin.index %% cumlevels[i]
        }
level.mat = level.mat + 1
out.df = data.frame(counts, level.mat)
dimnames(out.df) = list(1:length(counts),
                        names(A.df)[c(1, margin+1)])
out.df
}


#############################################################################

compact.data.frame = function(data.df){
# compacts table by combining identical factor level combinations
# expects counts in the first column

if(!is.numeric(data.df[,1]))stop(
                "data frame must have counts in the first column")
max.levels = get.dim(data.df)
prod = 1
index = 0
dimnames = vector(length=length(max.levels), mode="list")
for(i in 1:length(max.levels)){
  index = index + (as.numeric(unclass(data.df[,i+1]))-1)*prod
  prod = prod * max.levels[i]
  dimnames[[i]]=levels(data.df[,i+1])
}
totals = tapply(data.df[,1], index,sum)
X = matrix(0, length(totals),length(max.levels))
cumlevels = (c(1,cumprod(max.levels)[-length(max.levels)]))
index = as.numeric(names(totals))
for(i in length(max.levels):1){
X[,i] = index%/%cumlevels[i]
index = index %% cumlevels[i]
}
X=X+1
out.df = data.frame(Count=totals)
for(j in 1:length(max.levels))out.df =
data.frame(out.df,factor(X[,j], labels = levels(data.df[,j+1])))
```

```
colnames(out.df)=colnames(data.df)
out.df
}


#########################################################
# Function to compute G-squared for mixture model fit


G.sq = function(A, params){
# calculates G-squared
# A: probability array being fitted
# params: parameters retirned by function mixture.fitter.single
theta=params$theta
tau=params$tau
T=length(tau)
prob1=array(0, dim(A))
J = length(dim(A))
for(t in 1:T){
myprod=theta[[1]][,t]
if(J>1){for(j in 2:J)myprod=outer(myprod,theta[[j]][,t])}
prob1 = prob1 + tau[t]*myprod
}

sum(A*log(A/(prob1)))
}

#########################################################
pareto = function(A.df, params, number = 30,
      main="Fitted probabilities versus Frequencies"){

# compares fitted and empirical probs via a pareto graph

theta=params$theta
tau=params$tau
T=length(tau)
prob1=numeric(dim(A.df)[1])
J = dim(A.df)[2]-1
for(t in 1:T){
myprod=1
for(j in 1:J)myprod=myprod * theta[[j]][A.df[,j+1],t]
prob1 = prob1 + tau[t]*myprod
}

prob.empirical = A.df[,1]
prob.empirical = prob.empirical/sum(prob.empirical)

p.e = sort(prob.empirical, decreasing=TRUE)[1:number]
```

```
p.f = prob1[order(prob.empirical, decreasing=TRUE)][1:number]

barplot(rbind(p.e, p.f), beside=TRUE, legend.text=
c("Empirical probs", "Fitted probs"), main=main)

}

#########################################################
#

number.of.params = function(A.df,T, type="n"){
K = if(type=="n")get.dim.num(A.df) else get.dim(A.df)
T-1 + T*(sum(K)-length(K))
}

#########################################################

#########################################################
G.sq.log.lin = function(A.df, glm.obj){
# calculates log-likelihood
probs= predict(glm.obj, type="response")
probs=probs/sum(probs)
n=sum(A.df[,1])
sum(A.df[,1]*log(A.df[,1]/(n*probs)))
}


#########################################################
pareto.log.lin = function(A.df, glm.obj, number = 30,
        main="Fitted probabilities versus Frequencies"){

# compares fitted and empirical probs via a pareto graph

prob.fitted = predict(glm.obj, type="response")
prob.fitted = prob.fitted/sum(prob.fitted)

prob.empirical = A.df[,1]
prob.empirical = prob.empirical/sum(prob.empirical)

p.e = sort(prob.empirical, decreasing=TRUE)[1:number]
p.f = prob.fitted[order(prob.empirical,
                                decreasing=TRUE)][1:number]

barplot(rbind(p.e, p.f), beside=TRUE,
legend.text = c("Empirical probs", "Fitted probs"), main=main)

}
```

```
get.index = function(n, base.vec,j){

# converts the integer n into a mixed-base representation
# and returns the jth element of the representation
# not called by users

# eg  if (n-1) = i-1 + (j-1)*I + (k-1)*I*J + (l-1)*I*J*K
# returns (i-1,j-1,k-1,l-1)

Nvar=length(base.vec)
cumlevels = (c(1,cumprod(base.vec)[-Nvar]))
nn=n-1
for(index in length(base.vec):j){
l.index = nn%/%cumlevels[index]
nn = nn %% cumlevels[index]
}
l.index
}
#####################################################################

df2table.mix = function(data.df, tau=0.99){

# function to mix empirical distribution with independence
# distribution
# data.df: a data frame in standard form representing a table
#tau: the mixing probability

if(tau<0 || tau>1)stop("tau must be a probability")
# check if data.df is a proper representation of a table
if(!is.standard(data.df)) stop("First argument
                 must be a data frame in standard form")
table.e = df2table(data.df)
N=sum(table.e)
table.e = table.e/N
# compute independence table
table.i = tapply(data.df[,1], data.df[,2], sum)/N

if(dim(data.df)[2]>2)for(i in 3:dim(data.df)[2]){
  table.i = outer(table.i,  tapply(data.df[,1],
                              data.df[,i], sum)/N)
}

tau*table.e + (1-tau)*table.i
}
```

```
##########################################################
log.lin.fitter = function(countfile, Kvec,  MAXITER=100,
                                              TOL=1.0e-6){

# fits log-linear model counts ~ (A1....+AK)^2 to data
# in countfile
# countfile contains the counts in reverse lexographic order
# (first factor varying most rapidly)
# processes input file a row at a time
# length of row limited by computer memory
# Kvec is the vector of factor level lengths

if((MAXITER<1)&(round(MAXITER)!=MAXITER))stop(
"MAXITER must be a positive integer")

if(!is.dimvec(Kvec))stop("Kvec must be a vector of positive integers")


n.param = param.count(Kvec, order=2)  # currently only does 2 factor ints
beta =get.beta.start(countfile, Kvec) # get starting values
K=length(Kvec)


del=2*TOL
iter=1
while((iter<MAXITER)&&(del>TOL)){

  rhs = rep(0,n.param)
  A = matrix(0,n.param,n.param)
  i=1
  skip=0

  while(TRUE){
    y = scan(countfile, skip=skip, nlines = 1, quiet=TRUE)
    if(length(y)==0) break
    for(j in 1:length(y)){
      # get x
      ivec=n.to.nvec(i, Kvec) + 1
      x=1
      for(l in 1:K) x=c(x,(ivec[l]==(2:Kvec[l])))
      for(l1 in 1:(K-1)){
        for(l2 in (l1+1):K){
          dvec1 = (ivec[l1]==(2:Kvec[l1]))*1
          dvec2 = (ivec[l2]==(2:Kvec[l2]))*1
          x = c(x, as.vector(outer(dvec1,dvec2)))
      }}
```

```
      mu = exp(sum(x*beta))
      rhs = rhs + x*(y[j]-mu)
      A = A + outer(x,x)*mu
      i = i + 1
    }
    skip=skip + 1
  }
  delvec = solve(A,rhs)
  beta = beta + delvec
  del = sum(abs(delvec))
}
beta
}


############################################################
get.beta.start= function(countfile, Kvec){

# fits linear model log(counts) ~ (A1....+AK)^2 to data
# in countfile to give starting values for the function
# log.lin.fitter
# countfile contains the counts in reverse lexographic order
# (first factor varying most rapidly)
# length of rows of file limited by computer memory
# Kvec is the vector of factor level lengths

# currently only does 2 factor ints
n.param = param.count(Kvec, order=2)

K=length(Kvec)

  i=1
  rhs = rep(0,n.param)
  A = matrix(0,n.param,n.param)
  skip=0

  while(TRUE){
    y = scan(countfile, skip=skip, nlines = 1, quiet=TRUE)
    if(length(y)==0) break
    ly=ifelse(y==0, log(0.5), log(y))
    for(j in 1:length(y)){
      # get x
      ivec=n.to.nvec(i, Kvec) + 1
      x=1
      for(l in 1:K) x=c(x,(ivec[l]==(2:Kvec[l])))
      for(l1 in 1:(K-1)){
        for(l2 in (l1+1):K){
          dvec1 = (ivec[l1]==(2:Kvec[l1]))*1
```

```
          dvec2 = (ivec[l2]==(2:Kvec[l2]))*1
          x = c(x, as.vector(outer(dvec1,dvec2)))
      }}
      rhs = rhs + x*ly[j]
      A = A + outer(x,x)
      i = i + 1
    }
    skip=skip + 1
  }
solve(A,rhs)
}

############################################################
calculate.probs= function(beta, Kvec){

# generates fitted probabilities from a given parameter
# vector beta
# Kvec is the vector of factor level lengths

K=length(Kvec)

if(param.count(Kvec,2)!=length(beta))stop(
                       "beta incompatible with Kvec")
probs = numeric(prod(Kvec))
for(i in 1:prod(Kvec)){
      # get x
      ivec=n.to.nvec(i, Kvec) + 1
      x=1
      for(l in 1:K) x=c(x,(ivec[l]==(2:Kvec[l])))
      for(l1 in 1:(K-1)){
        for(l2 in (l1+1):K){
          dvec1 = (ivec[l1]==(2:Kvec[l1]))*1
          dvec2 = (ivec[l2]==(2:Kvec[l2]))*1
          x = c(x, as.vector(outer(dvec1,dvec2)))
      }}
      probs[i] = exp(sum(x*beta))
}
probs/sum(probs)
}

################################################
# fits  mixture models using ML and EM
# uses single array input, minimises storage requirements

# data structures

# Input data: a J-dimensional array A, containing the
# probabilities to be fitted
```

```
# The mixture will have T components

mixture.fitter.single = function(A, T, params0=NULL,
                         MAXITER=100, TOL=1.0e-8, trace=TRUE){

if(!is.probarray(A))stop(
       "First argument must be a probability array")
if((T<0)||(round(T)!=T))stop(
        "Second argument must be a positive integer")
if((MAXITER<1)&(round(MAXITER)!=MAXITER))stop(
"Argument 'MAXITER' must be a positive integer")
if(TOL<0)stop("Argument 'tol' must be positive")

# extract counts and totals


# get levels for all the factors


K = dim(A)
J = length(K)
N=prod(K)

if(is.null(params0)){
tau = runif(T)
tau=tau/sum(tau)

theta = vector(mode="list", length=J)
for(j in 1:(J)){
  theta[[j]] = matrix(runif(K[j]*T), K[j], T)
  theta[[j]] = t(t(theta[[j]]) /apply(theta[[j]], 2,sum))
}
} else{

tau=params0$tau
theta=params0$theta
}


iter=1
del=2*TOL

# now iterate
while((iter<MAXITER)&&(del>TOL)){
oldtau=tau
  # compute pt.tot
  p.tot = 0
```

```
  for(t in 1:T){
  ptt = theta[[1]][,t]
  if (J>1)for(j in 2:J)ptt  = outer(ptt, theta[[j]][,t])
  p.tot = p.tot + tau[t]*ptt
  }

  for(t in 1:T){
    ptt=theta[[1]][,t]
    if (J>1)for(j in 2:J)ptt  = outer(ptt, theta[[j]][,t])
    ptt = tau[t]*ptt/p.tot
    tau[t] = sum(A*ptt)

# compute thetas
    r1=1
    r2=N
    for(j in 1:J){
      r2 = r2/K[j]
      index = rep(rep(1:K[j],rep(r1,K[j])),r2)
      r1=r1*K[j]
      theta[[j]][,t] = my.tapply(A*ptt, index, K[j])/tau[t]
      }
    }
  del = sum(abs(tau-oldtau))
  if(trace)print(c(iter, del))
  iter=iter+1
  }

list(tau=tau, theta=theta, dimnames=dimnames(A))

}


############################################################
params2table = function(params){
# calculates probability table from mixture parameters

if(!is.params(params))stop(
             "Argument must be a set of mixture paramsters")



J = length(params$theta)
T = length(params$tau)
prob.table=0
for(t in 1:T){
  temp.table = params$theta[[1]][,t]
  if(J>1) for(j in 2:J) temp.table = outer(
                  temp.table, params$theta[[j]][,t])
```

```
  prob.table = prob.table + params$tau[t]*temp.table
  }
if(!is.null(params$dimnames))dimnames(prob.table) =
                                        params$dimnames
prob.table
}


##########################################################
my.tapply = function(x,g, ng){
# used in  ML.fitter.big.v2
sums = numeric(ng)
for(i in 1:length(x)) sums[g[i]] =
                                      sums[g[i]] + x[i]
sums
}


################################################
# fitting for mixture models using EM
# uses array input, minimises storage requirements

# data structures

# Input data: array.list, a list of arrays containing the
# probability tables to be to be fitted

# The mixture will have T components

mixture.fitter.fusion = function(array.list, T, params0=NULL,
                                     MAXITER=100, TOL=1.0e-8, trace=TRUE){

if((T<0)||(round(T)!=T))stop("Second argument must be a positive integer")

if(!is.list(array.list))stop(
      "First argument must be a list of relative frequency tables")


for(i in 1:length(array.list)){
 if(!is.probarray(array.list[[i]])) stop(
"First argument must be a list of relative frequency tables")
}

if((T<0)||(round(T)!=T))stop(
        "Second argument must be a positive integer")

if((MAXITER<1)&(round(MAXITER)!=MAXITER))stop(
"Argument 'MAXITER' must be a positive integer")
```

```
if(TOL<0)stop("Argument 'TOL' must be positive")

R = length(array.list)
# get names and dimensions
names.list = vector(mode="list", length=R)
dim.list = vector(mode="list", length=R)


vars=NULL; dims = NULL

for(i in 1:R) {
names.list[[i]] = names(dimnames(array.list[[i]]))
dim.list[[i]] = sapply(dimnames(array.list[[i]]), length)
vars = c(vars, names.list[[i]])
dims = c(dims, dim.list[[i]])
}

# remove duplicates and sort variable names into order

dup<-duplicated(vars)

var.names<-vars[!dup]
max.levels<-dims[!dup]

var.order = order(var.names)
var.names = var.names[var.order]
max.levels = max.levels[var.order]

# check for consistency of levels, record factor levels

J=length(var.names)  # J is number of variables

levels.list = vector(mode="list", length=J)
for(j in 1:length(var.names)){
  first=TRUE
  name = var.names[j]
  for(i in 1:length(array.list)){
    k = match(name, names(dimnames(array.list[[i]])))
    if( !is.na(k)){
       current.levels = dimnames(array.list[[i]])[[k]]
       if(first) { first.levels = current.levels
       first = FALSE} else
         {
         if(length(first.levels) !=  length(current.levels)) stop(
                paste("Factor levels for factor",name, "not compatible"))
            if(any(first.levels!= current.levels)) stop(
                paste("Factor levels for factor",name, "not compatible"))
```

```
         }

       }

   }

levels.list[[j]] = first.levels
}


names(levels.list) = var.names
varmat = matrix(FALSE, J, R)
for( i in 1: R) varmat[match(names(dimnames(array.list[[i]])),
                  var.names),i]=TRUE


# get levels for all the factors

if(is.null(params0)){
# set up and initialise data structures for the parameters

tau = runif(T)
tau=tau/sum(tau)

theta = vector(mode="list", length=J)
for(j in 1:J){
theta[[j]] = matrix(runif(max.levels[j]*T),
                    max.levels[j], T)
theta[[j]] = t(t(theta[[j]])/apply(theta[[j]], 2,sum))
}
}else
{
tau=params0$tau
theta=params0$theta
}

ST =  vector(mode="list", length=T)

iter=1
del=2*TOL

# now iterate
while((iter<MAXITER)&&(del>TOL)){
oldtau=tau
oldtheta = theta
# compute pt.tot
for (t in 1:T){
  S=0
```

```
    for(j in 1:J)ST[[j]] = numeric(max.levels[j])
    for(r in 1:R){
      jvec = (1:J)[varmat[,r]]
      # compute denom of Q[i1,...ir,t]
      p.tot = 0
      for(tt in 1:T){
        ptt = oldtheta[[jvec[1]]][,tt]
        if (length(jvec)>1)for(j in 2:length(jvec))ptt  =
                            outer(ptt, oldtheta[[jvec[j]]][,tt])
        p.tot = p.tot + oldtau[tt]*ptt
       }

    # compute Q[i1,...ir,t]
     ptt = oldtheta[[jvec[1]]][,t]
     if (length(jvec)>1)for(j in 2:length(jvec))ptt =
                               outer(ptt, oldtheta[[jvec[j]]][,t])
     ptt = oldtau[t]*ptt/p.tot
   # Accumulate sums for sum(r) f[i1,...ir]Q[i1,...ir,t]
     S = S + sum(array.list[[r]]*ptt)

   # accumulate sums for sum(j,l,r) f[i1,...ir]Q[i1,...ir,t]
     r1=1
     K = dim.list[[r]]
     r2=prod(K)
     for(j in 1:length(jvec)){
       r2 = r2/K[j]
       index = rep(rep(1:K[j],rep(r1,K[j])),r2)
       r1=r1*K[j]
       ST[[jvec[j]]] = ST[[jvec[j]]] +
                       my.tapply(array.list[[r]]*ptt, index, K[j])
       }

    }
    tau[t] = S/R
    for(j in 1:J) theta[[j]][,t] = ST[[j]]/sum(ST[[j]])

  }
  del = sum(abs(tau-oldtau))
  iter=iter+1
  if(trace)print(c(iter,del))
}


list(tau=tau, theta=theta, dimnames=levels.list)

}

#################################################################
```

```
# functions for generating data

#################################################################
# generate.sample.prob

generate.sample.prob = function(N, file, prob.table){

# generates a csv file containing the a sample generated according
# to the probabilities in  prob.table

if(!is.probarray(prob.table))stop(
      "last argument must be a relative frequency table")

if((N<0)&(round(N)!=N))stop(
"First argument must be a positive integer")



index = sample(prod(dim(prob.table)), N, replace=TRUE,
        prob = prob.table)

X = matrix(0,N,length(dim(prob.table)))
for(i in 1:length(index)) X[i,]=n.to.nvec(index[i], dim(prob.table))+1

X.df=data.frame(X)
colnames(X.df)  = if(is.null(names(dimnames(prob.table)))) {
         paste("V",1:length(dim(prob.table)), sep="")} else
         names(dimnames(prob.table))
write.table(X.df, file, row.names=FALSE, quote=FALSE, sep=",")
}

#################################################################
# generate.sample.MH

generate.sample.MH = function(N, file, beta,
margin.list, thin=10, burn = 1000){

if((N<0)&(round(N)!=N))stop(
"First argument must be a positive integer")

# generates a csv file containing a sample generated
# according to the log-linear parameters in param.vec

Kvec = unlist(lapply(margin.list, length))
X = generate.data.MH(beta, margin.list, Kvec, N,
thin=thin, burn = burn)
X.df=data.frame(X)
colnames(X.df)  = if(is.null(names(margin.list))){
```

```
 paste("V",1:length(Kvec), sep="")} else names(margin.list)
write.table(X.df, file, row.names=FALSE, quote=FALSE, sep=",")
}


#####################################################################
generate.sample.mix = function(N, file, params){

if(!is.params(params))stop(
      "last argument must be a set of mixture parameters")

if((N<0)&(round(N)!=N))stop(
"First argument must be a positive integer")

J = length(params$theta)

T = length(params$tau)
X =NULL

tvec = as.vector(rmultinom(1,  N,  prob = params$tau))
for( t in 1:T){
X.temp = matrix(0,tvec[t], J)
for(j in 1:J) X.temp[,j] = sample(dim(params$theta[[j]])[1], tvec[t],
                           replace=TRUE, prob = params$theta[[j]][,t])
X = rbind(X,X.temp)
}
# randomly reorder rows

X = X[order(runif(N)),]
X.df=data.frame(X)
colnames(X.df)  = if(is.null(names(params$dimnames))) {
     paste("V",1:length(Kvec), sep="")} else names(params$dimnames)
write.table(X.df, file, row.names=FALSE, quote=FALSE, sep=",")
}



#####################################################################
make.margin = function(countfile, Kvec){

if(!is.dimvec(Kvec))stop(
       "Second argument must be a vector of positive integers")

 K = length(Kvec)
  margin.list = vector(length=K, mode="list")
  for(k in 1:K) margin.list[[k]]  = rep(0, Kvec[k])
  i=1
  skip=0
```

```
  while(TRUE){
    y = scan(countfile, skip=skip, nlines = 1, quiet=TRUE)
    if(length(y)==0) break
    for(j in 1:length(y)){
      ivec=n.to.nvec(i, Kvec) + 1
      for(k in 1:K) margin.list[[k]] [ivec[k]] =
                        margin.list[[k]] [ivec[k]] + y[j]
      i = i + 1
      }
    skip=skip + 1
    }
margin.list
}
#################################################################
# program to generate data using the Metropolis-Hastings algorithm

generate.data.MH = function(param.vec, margin.list, Kvec, N,
 thin=10, burn = 1000){

# generates data using  using the Metropolis-Hastings algorithm
# param.vec  is a parameter vector produced by the function
# log.lin.fitter
# Kvec gives the number of levels in the factors
# thin is the thinning period, run chain for thin*N
# values to return N
# burn is the length of the burn-in period
# margin.list is the list of marginal distributions

if(!is.dimvec(Kvec))stop(
      "Kvec must be a vector of positive integers")

if((N<0)&(round(N)!=N))stop(
"N must be a positive integer")


get.random = function(margin.list){
i = numeric(length(margin.list))
for(k in 1:length(margin.list)){
  i[k] = sample(length(margin.list[[k]]),
         1, prob=margin.list[[k]])
  }
i
}


get.prob = function(i,margin.list){
prob = 1
for(k in 1:length(margin.list)) prob = prob *
```

```
                          (margin.list[[k]])[i[k]]
prob
}

get.x = function(i, Kvec){
K=length(Kvec)
x=1
for(l in 1:K) x=c(x,(i[l]==(2:Kvec[l])))
    for(l1 in 1:(K-1)){
      for(l2 in (l1+1):K){
         dvec1 = (i[l1]==(2:Kvec[l1]))*1
         dvec2 = (i[l2]==(2:Kvec[l2]))*1
         x = c(x, as.vector(outer(dvec1,dvec2)))
     }}
x
}

# store results in matrix X

X = matrix(0, N, length(Kvec))
index = 1
# initial value

i = get.random(margin.list)
q.i = get.prob(i,margin.list)
xi = get.x(i,Kvec)
pi.i = exp(sum(xi*param.vec))

n = burn + thin*N
for (iter in 1:n){
j = get.random(margin.list)
q.j = get.prob(j,margin.list)
Xj = get.x(j,Kvec)
pi.j = exp(sum(Xj*param.vec))
alpha = min(1, (pi.j*q.j)/(pi.i*q.i))
if(runif(1)<alpha) {
i=j
q.i = get.prob(i,margin.list)
Xi = get.x(i,Kvec)
pi.i = exp(sum(Xi*param.vec))
}
if ((iter>burn)&&((iter-burn)%%thin)==0){
X[index,] = i
index = index + 1
}
print(iter)
}
X
```

```
}

#################################################################
# function to make all the data frames
# input is null or a directory path

make.all = function(...){
input.list<-list(...)
path = if(length(input.list)==0) getwd() else input.list[[1]]

path = gsub("\\","/",path, fixed=TRUE)

path.parts=unlist(strsplit(path,"/"))
if(rev(path.parts)[1]!="SNZTables")stop(
                                "Directory must be SNZtables")
current.dir = getwd()

# save directory

path1 = paste(path,"usually resident", sep="/")
setwd(path1)
source("birthplace.sex\\birthplace.sex.r")
source("languages.age.sex\\languages.age.sex.r")
source("yearsatres.age.sex\\yearsatres.age.sex.r")

setwd(current.dir)


path2 = paste(path,"usually resident 15+", sep="/")
setwd(path2)

files =  list.files()
r.files = paste(files,"\\",files,".r", sep="")
for(i in 1:length(files))source(r.files[i])


setwd(current.dir)


path3 = paste(path,"employed usually resident 15+", sep="/")
setwd(path3)

files = list.files()
r.files = paste(files,"\\",files,".r", sep="")
for(i in 1:length(files))source(r.files[i])

# restore directory
invisible(setwd(current.dir))
```

```
}


#####################################################
KLDist = function(my.array,params){

# calculates KL distance between distn in my.array
# and mixture distribution

if(!is.probarray(my.array)) stop("(First argument must be
                  a probability array")
if(!is.params(params)) stop(
"Second argument must be a set of mixture parameters")

my.table = params2table(params)
sum(my.array*log(my.array/my.table))
}

#####################################################
loglin2table= function(beta, Kvec, dimnames=NULL){

# generates fitted probabilities from a given log-linear
# parameter vector beta for 2fi model
# Kvec is the vector of factor level lengths

K=length(Kvec)
if(param.count(Kvec,2)!=length(beta))stop(
                          "beta incompatible with Kvec")
probs = numeric(prod(Kvec))
for(i in 1:prod(Kvec)){
      # get x
      ivec=n.to.nvec(i, Kvec) + 1
      x=1
      for(l in 1:K) x=c(x,(ivec[l]==(2:Kvec[l])))
      for(l1 in 1:(K-1)){
        for(l2 in (l1+1):K){
          dvec1 = (ivec[l1]==(2:Kvec[l1]))*1
          dvec2 = (ivec[l2]==(2:Kvec[l2]))*1
          x = c(x, as.vector(outer(dvec1,dvec2)))
      }}
      probs[i] = exp(sum(x*beta))
}
array(probs/sum(probs), Kvec, dimnames)
}

#####################################################
# functions for error checking
```

```
is.params = function(params){
if(!is.list(params))return(FALSE)
if(!is.prob(params$tau))return(FALSE)
if(!is.list(params$theta))return(FALSE)
for(i in 1:length(params$theta))if(!is.probmat(params$theta[[i]]))return(FALSE)
TRUE
}

is.prob = function(x){
if(!is.vector(x))return(FALSE)
if(any(x<0))return(FALSE)
if(abs(sum(x)-1)>1.0e-10)return(FALSE)
TRUE
}


is.probmat = function(X){
if(!is.matrix(X))return(FALSE)
for(j in 1:ncol(X))if(!is.prob(X[,j]))return(FALSE)
TRUE
}

is.probarray = function(X){
if(!is.array(X))return(FALSE)
if(any(X<0))return(FALSE)
if(abs(sum(X)-1)>1.0e-10)return(FALSE)
TRUE
}


is.dimvec = function(x){
 if(!is.vector(x))return(FALSE)
 if(any(x<0))return(FALSE)
 if(any(x!=round(x)))return(FALSE)
TRUE
}
```