UCS/Perl Documentation

Stefan Evert

printed on June 4, 2006

Contents

1 General Documentation

1.1 ucsintro

A first introduction to UCS/Perl

INTRODUCTION

UCS is a set of libraries and tools intended for the empirical study of cooccurrence statistics. Its major uses are to *apply* such statistics, called **association measures**, to cooccurrence data obtained from a corpus, and to *evaluate* the resulting association scores and rankings against (manually annotated) reference data.

The frequency data extracted from a given corpus for a given type of cooccurrences consists of a list of **pair types** with their **frequency signatures** (i.e. joint and marginal frequencies), and is referred to as a **data set**. See (Evert 2004) for a detailed explanation of these concepts, different types of cooccurrences, and correct methods for obtaining frequency data. Data sets, stored in a special **.ds** file format, are the fundamental objects of the UCS toolkit. Most UCS programs manipulate or display such data set files.

The UCS implementation relies heavily on the programming language **Perl** (*http://www.perl.com/*) and the free statistical environment **R** (*http://www.r-project.org/*) as a library of mathematical and statistical functions. The core of UCS is written in Perl (the **UCS/Perl** part), but there is also a small library of R functions for interactive work within R (the **UCS/R** part). UCS/Perl uses R as a back-end, making the most important statistical functions available through a Perl module.

UCS/Perl is mainly a collection of Perl modules that perform the following tasks:

- read and write data set files (.ds, .ds.gz)
- manage in-memory representations of data sets
- compile UCS expressions for easy access to data set variables
- filter, annotate, sort, and analyse data sets
- provide a repository of built-in association measures
- display data sets and evaluation graphs (Perl/Tk and R) [**not implemented yet**]

Most UCS programs will be custom-built scripts, using the library of support functions provided by the UCS/Perl modules. Loading a data set, annotating it with association scores from one or more measures, and sorting it in various ways can be done with a few lines of Perl code. There are also some ready-made programs in UCS/Perl that perform such standard tasks, operating on data set files. A substantial part of the UCS/Perl functionality is thus accessible from the command-line, at the cost of some additional overhead compared to a custom script (which operates on in-memory representations).

Below, you will find a list of the general documentation files, Perl modules, and programs that are included in the UCS/Perl distribution. Manpages for all modules and programs (as well as the general documentation) are easily accessible with the **ucsdoc** program, and can also be formatted for printing.

General Documents

```
ucsdoc ucsintro             # this introduction
ucsdoc ucsfile              # description of the UCS data set file format (.ds)
ucsdoc ucsexp               # UCS expressions and wildcards
ucsdoc ucsam                # overview of built-in association measures
```

UCS/Perl MODULES

```
use UCS;                    # core library
use UCS::File;              # file access utilities
use UCS::R;                 # interface to UCS/R
use UCS::SFunc;             # special functions and statistical distributions

use UCS::Expression;        # Perl code interspersed with UCS variables
use UCS::Expression::Func;  # utility functions available in UCS expressions

use UCS::AM;                # implementations of various association measures
use UCS::AM::HTest;         # add-on package: variants of hypothesis tests
use UCS::AM::Parametric;    # add-on package: parametric association measures

use UCS::DS;                # data sets ...
use UCS::DS::Stream;        #   i/o streams for data set files
use UCS::DS::Memory;        #   in-memory representation of data sets
use UCS::DS::Format;        #   ASCII formatter (+ other formats)
```

See the respective manpages (`ucsdoc ModuleName`) for more information.

UCS/Perl PROGRAMS

```
ucsdoc          # front-end to perldoc
ucs-config      # automatic configuration of UCS/Perl scripts
ucs-tool        # find and run user-contributed UCS/Perl scripts
ucs-list-am     # list built-in association measures & add-on packages

ucs-make-tables # compute frequency signatures from list of pair tokens
ucs-summarize   # print (statistical) summaries for selected variables

ucs-select      # select rows and/or columns from a data set file
ucs-add         # add variables to a data set file
ucs-join        # combine rows and/or columns from two data sets
ucs-sort        # sort data set file by specified attribute(s)

ucs-info        # display information from header of data set file
ucs-print       # format data set as ASCII table (for viewing and printing)
```

See the respective manpages (`ucsdoc ProgramName`) for more information.

TRIVIA

UCS stands for **Utilities for Cooccurrence Statistics**.

REFERENCES

Evert, Stefan (2004). *The Statistics of Word Cooccurrences: Word Pairs and Collocations.* PhD Thesis, University of Stuttgart, Germany.

On-line repository of association measures: *http://www.collocations.de/*

COPYRIGHT

1.2 ucsfile

The UCS data set file format

INTRODUCTION

UCS data sets are stored in a simple tabular format, similar to that of a statistical table. Each row in the table corresponds to a **pair type**, and its individual fields (columns) provide various kinds of information about the pair type:

- a unique **ID number** (unique within the data set)
- the component **lexemes**
- the pair type's **frequency signature**
- [optional] contingency tables of **observed** and **expected frequencies** computed from the frequency signature
- [optional] **coordinates** computed from the frequency signature
- **association scores** and **rankings** for various association measures
- arbitrary **user-defined attributes**, especially for the manual annotation of *true positives* in an evaluation study

Following statistical terminology, the table columns are referred to as the **variables** of a data set (each of which assumes a specific value for each pair type). Columns are separated by a TAB character (`"\t"`), and the first row lists the **variable names** as table headings (see the section §1.2 below for naming conventions).

The actual data table may be preceded by an optional **header** of Perl-style comment lines (beginning with a `#` character). Lines with the special format

```
##:: <variable> = <value>
```

define **global variables**, which may be interpreted by some of the UCS/Perl programs (see the section §1.2 below). The variable name (*variable*) may only contain alphanumeric characters (`A-Z a-z 0-9`) and the period (`.`). The *value* may contain arbitrary characters, including whitespace (but leading and trailing whitespace will be ignored). Variable definitions must not span multiple lines.

UCS data set files must have the filename extension **.ds**. They may be compressed with **gzip** (and they usually are), in which case they carry the extension **.ds.gz**. UCS library functions will automatically recognise and uncompress data set files with this extension.

A special subtype of data sets are the **annotation database** files with extension **.adb** (uncompressed) or **.adb.gz** (compressed). Annotation databases omit all frequency information and association scores, listing only component lexemes and user-defined attributes. They are used as repositories of lexical information (such as manually annotated *true positives* for evaluation purposes) that applies to data sets extracted from different corpora (or with different methods).

GLOBAL VARIABLES

```
size        number of pair types in a data set
```

The only global variable that is currently supported is **size**, an integer specifying the number of pair types in a data set. Availability of the data set size in the header may give a slight performance improvement when loading data set files into memory. If **size** is set to an incorrect value, the behaviour of UCS/Perl programs and modules is undefined.

A global variable whose name is identical to that of a variable defined in the data set (i.e. a table column) is interpreted as an **explanatory note**. Such notes should typically be given for all user-defined variables, and also for user-defined association measures.

Unsupported variables will simply be ignored and will not raise errors or warnings when a data set file is parsed.

DATA TYPES

The UCS system supports four different data types:

```
BOOL      a logical (Boolean) value
INT       a signed integer value (>= 32 bits)
DOUBLE    a floating-point value (IEEE double precision)
STRING    an arbitrary string (ISO-8859-1 or UTF-8)
```

Boolean values are represented by 1 (true) and 0 (false). **String** values may contain blanks (but no TAB characters) and are neither quoted nor escaped. Full support for Unicode strings (UTF-8) is only available within the UCS/Perl subsystem.

The UCS/R subsystem will interpret Boolean values as logical variables, and strings (except for the component lexemes) as *factor* variables with a fixed set of levels (which are automatically determined from the data).

User-defined attributes may assume the special value `NA` for **missing values**. (Note that the string `NA` will always be interpreted as a missing value rather than a literal character string!) UCS/R has built-in support for missing values, whereas UCS/Perl represents them by **undef** entries. Programs that do not support missing values may replace them by 0 (BOOL and INT), 0.0 (DOUBLE), or the empty string "" (STRING).

The **data type** of a variable is uniquely determined by the variable name, as detailed in the section §1.2 below.

VARIABLES

In order to be compatible with the **R** language, variable names may only contain alphanumeric characters (`A-Z a-z 0-9`) and periods (`.`), and they must begin with a letter. The main function of periods is to delimit words in complex variable names, replacing blanks, hyphens, and underscores. UCS variable names are case-sensitive.

Periods are not allowed in **Perl** variable names, but **UCS expressions** provide a special syntax for direct access to data set variables (see the ucsexp and UCS::Expression manpages). In the rare case where plain Perl variables are used to store information from a data set, periods should be replaced by underscores (_) in the variable names.

There are strict **naming conventions** for data set variables, which are detailed in the following subsections. Apart from a fixed list of core variables (whose names do not contain the `.` character), all variable names begin with a period-separated **prefix** that determines the data type of the variable.

Core Variables Core variables represent the minimal amount of information that must be present in a data set file (i.e. evidence for cooccurrences extracted from a corpus). All core variables are mandatory, except in the case of annotation database files (.adb), which omit frequency signatures (`f f1 f2 N`). For relational cooccurrences, frequency signatures can be computed with the **ucs-make-tables** utility from a stream of pair tokens (cf. the ucs-make-tables manpage).

```
INT    id    a numerical ID value (unique within the data set)
STRING l1    first component type of the pair
STRING l2    second component type of the pair

INT    f     cooccurrence frequency of pair type
INT    f1    marginal frequency of first component
INT    f2    marginal frequency of second component
INT    N     sample size (identical for all pair types)
```

`id` is a numerical ID value, which must be unique within a data set. Its intended uses are to identify pair types in subsets selected from a given data set, and to validate line numbers when attributes or association scores are computed by an external program and re-integrated into the data set file.

The **lexemes** `l1` and `l2` are the component (word) types that uniquely identify a pair type. Consequently, a data set file must not contain multiple rows with identical `l1` and `l2` values. UCS/Perl should provide reasonably good support for Unicode strings as lexemes (in UTF-8 encoding), at least when running on Perl version 5.8.0 or newer.

The quadruple `f f1 f2 N` is called the **frequency signature** of a pair type. It contains all the frequency information used by **association measures** and is equivalent to a contingency table. Note that the **sample size** `N` is identical for all pair types in a data set and is included here mainly for convenience' sake (so that association scores can be computed from the row data without reference to a global variable). See (Evert 2004) for more information on lexemes and frequency signatures.

Derived Variables Derived variables can be computed from the frequency signatures of pair types, providing different "views" of the frequency information. Normally, they are not annotated explicitly but are accessible through **UCS expressions**, which compute the required values automatically (see the ucsexp and UCS::Expression manpages).

```
INT    O11   contingency table of observed frequencies
INT    O12     (computed from frequency signature)
INT    O21
INT    O22

INT    R1    row sums in observed contingency table
INT    R2
INT    C1    column sums in observed contingency table
INT    C2
```

The variables `O11 O12 O21 O22` represent the observed **contingency table** of a pair type. Note that their frequency information is equivalent to the frequency signature of the pair type. In addition, the **row sums** (`R1 R2`) and **column sums** (`C1 C2`) of the contingency table are also made available.

```
DOUBLE E11   contingency table of expected frequencies
DOUBLE E12     under point null hypothesis
DOUBLE E21     (computed from row and column sums)
DOUBLE E22
```

The variables `E11 E12 E21 E22` represent the contingency table of **expected frequencies**, i.e. the expectations of the multinomial sampling distribution under the point null hypothesis of independence. Most association measures compare observed frequencies to expected frequencies in some way.

In a **geometric interpretation** of a data set, each pair type can be interpreted as a point x in a three-dimensional **coordinate space** $\boldsymbol{P}$. Since the sample size `N` is a constant parameter within the data set, the coordinates of x are given by the joint and marginal frequencies `f f1 f2`.

```
DOUBLE lf    logarithmic coordinates
DOUBLE lf1     (base 10 logarithm)
DOUBLE lf2
```

Since the coordinates usually have a skewed distribution across several orders of magnitude, it is often more convenient to visualise them on a logarithmic scale. The variables `lf lf1 lf2` give the **base ten logarithms** of the coordinate triple `f f1 f2`.

```
DOUBLE e     ebo-coordinates
DOUBLE b       (expected, balance, observed)
DOUBLE o

DOUBLE le    logarithmic ebo-coordinates
DOUBLE lb      (base 10 logarithm)
DOUBLE lo
```

Theoretical and empirical studies of the properties of association measures will often be based on transformed coordinate systems in the coordinate space. The most useful system are the **ebo-coordinates** `e b o` (for *expected*, *balance*, *observed*). All three coordinates range from 0 to infinity (constrained by the sample size parameter `N`). The base 10 logarithms `le lb lo` of the ebo-coordinates are convenient for visualisation purposes. `le` and `lb` range from -infinity to +infinity, while `lo` ranges from 0 to infinity (all constrained by `N`).

For backward compatibility, a transformation of the coordinate system to **relative frequencies**, which were used in earlier versions of this software, is also supported. The relative cooccurence (`p`) and marginal (`p1 p2`) frequencies are computed from the frequency signature according to the equations `p = f/N`, `p1 = f1/N`, and `p2 = f2/N`. Note that the logarithmic versions `lp lp1 lp2` are *negative* base 10 logarithms, ranging from 0 to infinity.

Association Scores and Rankings These variables store association scores and rankings for an arbitrary number of **association measures**. Each association measure is identified by a *key*, which is appended to the respective variable name prefix (resulting in the names `am.`*`key`* and `r.`*`key`*). See the UCS::AM manpage (and the manpages of the add-on packages listed there) for a wide range of built-in association measures.

```
DOUBLE am.*  association scores from measure identified by *
INT    r.*   ranking for this measure (ties are allowed)
```

Rankings are often computed on the fly, but they may also be annotated in data set files. Note that the `r.*` variables should *not* break ties but report identical ranks (and skip an appropriate number of subsequent ranks). The **ucs-sort** program (cf. the ucs-sort manpage) can be used to resolve ties in various ways (using other association scores, lexical sort order, or randomisation).

User-Defined Variables User-defined variables may contain arbitrary information, which is typically used for filtering data sets and to determine true positives in evaluation tasks. However, some special-purpose association measures may also base their association scores on their values. In order to allow a minimal amount of automatic processing (such as sorting by user-defined attributes), the variable name prefix of a user-defined variable is used to determine its data type, according to the following list.

```
BOOL   b.*   user-defined Boolean variable
INT    n.*   user-defined integer variable (n=number)
DOUBLE x.*   user-defined floating-point variable
STRING f.*   user-defined string variable (f=factor)
```

User-defined variables with the additional prefix `ucs` (corresponding to variable names `b.ucs.*`, `n.ucs.*`, `x.ucs.*`, and `f.ucs.*`) are reserved for internal use by UCS modules and programs.

REFERENCES

Evert, Stefan (2004). *The Statistics of Word Cooccurrences: Word Pairs and Collocations.* PhD Thesis, University of Stuttgart, Germany.

COPYRIGHT

1.3 ucsexp

Introduction to UCS expressions and wildcard patterns

INTRODUCTION

UCS expressions and wildcard patterns are two central features of the **UCS/Perl** system, which are to a large part responsible for its convenience and flexibility.

UCS **wildcard patterns** are used by most command-line tools to select data set variables with the help of shell-like wildcard characters (`?`, `*`, and `%`). A programmer interface is provided by the **UCS::Match** function from the **UCS** module (see the UCS manpage).

UCS expressions give easy access to data set variables from Perl code. With only a basic knowledge of Perl syntax, users can compute association scores and select rows from a data set (using the **ucs-add** and **ucs-select** utilities). The programmer interface is provided by the **UCS::Expression** module (see the UCS::Expression manpage for details). Before reading §1.3, you should become familiar with the UCS data set format and variable naming conventions as described in the ucsfile manpage.

When used on the **command line**, wildcard patterns usually have to be quoted to keep the shell from expanding wildcards (the GNU Bash shell knows better, though, unless there happen to be matching files in the current directory). Note that when a list of variable names and patterns is passed to one of the UCS/Perl utilities, each name or wildcard pattern has to be quoted *individually*. UCS expressions (almost) always have to be quoted on the command-line. Single quotes (`'...'`) are highly recommended to avoid interpolation of variables and other meta-characters. The UCS/Perl utilities expect a UCS expression to be passed as a single argument, so the expression must be written as one string. In particular, any expression containing whitespace must be quoted.

UCS WILDCARD PATTERNS

As described in the ucsfile manpage, UCS **variable names** may only contain the alphanumeric characters (`A-Z a-z 0-9`) and the period (`.`), which serves as a general-purpose word delimiter. There is a fixed set of **core variables**, whose names do not contain a period. All other variable names must begin with a prefix (one of `am. r. b. n. x. f.`) that determines the data type of the variable. The three **wildcard** characters take the special role of the period into account. Their meanings are

```
? ... a single character, except "."
* ... a string that does NOT contain a "."
% ... an arbitrary string of characters
```

The `%` wildcard is typically used to select variable names with a specific prefix or suffix, while `*` matches the individual words (or parts of words) in a complex variable name.

Examples

- a pattern without wildcard characters corresponds to a literal variable name : `id`, `O11`, `am.log.likelihood`
- the pattern `*` matches all core variables (and nothing else); `%` matches *all* variable names
- `O*` matches the derived variables *O11*, *O12*, *O21*, and *O22*; `*11` matches *O11* and *E11*, but no complex variable names
- prefix patterns allow us to select variables by their type, e.g. `am.%` for all association scores, or `f.%` for all user-defined string variables (factors); the `*` wildcard is inappropriate here because the variable names may contain additional period after the prefix

- when variable names are chosen systematically, prefix patterns can also be used to select meaningful groups of variables: `am.chi.squared%` matches all association scores that are derived from a chi-squared test, and `am.%.pv` matches all association scores that can be interpreted as probability values (see the UCS::AM and UCS::AM::HTest manpages for more information)

UCS EXPRESSIONS

An UCS expression consists of ordinary Perl code extended with a special syntax to access data set variables. This code is compiled on the fly and applied to the rows of a data set one at a time. The return value of a UCS expression is the value of the last statement executed, unless there is an explicit `return` statement. When the expression is used as a condition to select rows from a data set, it evaluates to true or false according to the usual Perl rules (the empty string `''` and the number `0` are false, everything else is true).

Data set variables are accessed by their variable name enclosed in `%` characters. They evaluate to the respective value for the current row in the data set and can be used like ordinary scalar variables in Perl. Thus, `%f%` corresponds to the cooccurrence frequency `f` of a pair type, `%l1%` and `%l2%` to its component lexemes, and `%am.log.likelihood%` to an association score from the log-likelihood measure. Derived variables (see the ucsfile manpage) do not have to be annotated explicitly in a data set. When necessary, they are computed on the fly from a pair type's frequency signature. Variable references should be treated as read-only (they are automatically localised so that assigning a new value to a UCS variable reference does not modify the original data set).

Any temporary variables needed by the Perl code should be made lexical by declaring them with the `my` keyword. Variable names beginning with an underscore (such as `$_f` or `$_n_total`) are reserved for internal use. Please don't use global variables, which pollute the namespaces and might interfere with other parts of the program. If you feel that you absolutely need a variable to carry information from one row to the next, use a fully qualified variable name in your own namespace.

Since a UCS expression is compiled by the Perl interpreter, it offers the full power and flexibility of Perl, but it also shares its idiosyncrasies and traps for the unwary. You should have a good working knowledge of Perl in order to write UCS expressions. If you don't know the difference between `==` and `eq`, now is the time to type `perldoc perl` and start reading the Perl documentation.

Just as in Perl, data types are automatically converted as necessary. Missing values (which appear as `NA` in data set files) are represented by **undef** in Perl. When there may be missing values in a data set, test for definedness (e.g. with `defined(%b.colloc%)`) to avoid warning messages. UCS expression can use all standard Perl functions (described on the perlfunc manpage). In addition, the utility functions from **UCS::Expression::Func** (see the UCS::Expression::Func manpage for a detailed description) and a range of special mathematical and statistical functions defined in the **UCS::SFunc** module (see the UCS::SFunc manpage for a complete listing and details) are imported automatically and can be used without qualification.

UCS Expressions for Programmers The programmer interface to UCS expressions is provided by the **UCS::Expression** module (see the UCS::Expression manpage), with functions for compiling and evaluating UCS expressions. The **UCS::DS::Memory** module includes several methods that apply a UCS expression to the in-memory representation of a UCS data set. Note that all built-in association measures are implemented as UCS expressions (see the UCS and UCS::AM manpages for more information, or have a look at the source files).

When you want to use external functions (either defined by your own module or imported from a separate module), they must be fully qualified. For instance, you must write `Math::Trig::atan(1)` instead of just `atan(1)`. Make sure that the module is loaded (with `use Math::Trig;`) before the expression is evaluated for the first time. You can just put the `use` statement in the Perl script or module where the UCS expression is defined, and it is probably also safe to include it in the expression itself (which allows you to use external libraries even in UCS expression typed on the command line).

An advanced feature of UCS expressions that is only available through the programmer interface are **parameters**. Parameters play the role of constants in UCS expressions: they can be accessed

like data set variables, but their values are fixed and stored within the **UCS::Expression** object. Parameter names must be valid UCS identifiers and should be all uppercase in order to avoid conflicts with variable names. Parameters must be declared and intialised when the UCS expression is compiled. Their values can be changed with the **set_param** method. See the UCS::Expression manpage for more information.

Examples

- The simplest UCS expressions compare the values of a data set variable to a constant. Recall that `==` is used for numerical comparison and `eq` for string comparison in Perl. Both operands will automatically be converted into an appropriate data type.

```
%f% == 1              # hapax legomena (single occurrences)

%f% >= 5              # pair types with cooccurrence freq. >= 5

%l1% eq "black"       # first component type is "black"
```

  Since UCS expressions are essentially short Perl scripts, the `#` character can be used to introduce line comments. String variables can also be matched against Perl regular expressions:

```
%l2% =~ /ness$/       # second component ends in ...ness
```

- Such simple comparisons can be combined into complex Boolean expressions. Use of the lexical operators `and`, `or`, and `not` is recommended for readability (and to avoid confusion with bit operators). Parentheses can also improve readability and help to avoid ambiguities.

```
%f% >= 5 and %f% < 10         # pair types in frequency range 5 .. 9

# pair types that are ranked high by t-score, but not by log-likelihood
(%r.t.score% <= 100) and not (%r.log.likelihood% <= 100)
```

- Missing values (`NA`) in a data set can be detected with Perl's **defined** operator. It may be useful to test data set variables before using them in order to avoid warning messages. The following examples assume a user-defined integer variable `n.accept`, which lists the number of annotators who have accepted a particular pair type as a collocation.

```
not defined(%n.accept%)     # selects rows where n.accept has the value NA

%n.accept% >= 1             # will print warnings for all NA values

defined(%n.accept%) and (%n.accept% >= 1)  # this is safe
```

- UCS expressions may contain multiple Perl statements, which must be separated by semicolon (`;`) characters. In this way, a complex formula can be broken down into smaller parts. The value of the expression is determined by the last statement (or by an explicit **return** command). Temporary variables that hold intermediate values should always be declared with lexical scope (using `my`). The first example computes the minimum of two frequency ratios, using the pre-declared `min()` function from **UCS::Expression::Func**.

```
# UCS expression may also extend over multiple lines
my $ratio1 = %f% / %f1%;
my $ratio2 = %f% / %f2%;
min($ratio1, $ratio2);      # min() is pre-declared
```

The second example shows how temporary variables can be used to replace missing values with defaults. Here the integer variable `n.accept` (for the number of annotators that accepted the given pair type as a collocation) defaults to 0.

```
my $n = (defined %n.accept%) ? %n.accept% : 0;
$n >= 1;
```

The third example identifies prime numbers used as `ID` values.

```
foreach my $x (2 .. int(sqrt(%id%))) {
  return 0 if (%id% % $x) == 0;
}
return 1;
```

Dirty Tricks Things *not* to do ...

- Global variables can be used to carry information from one row to the next (while lexicals will be re-instantiated and possibly initialised for each row they are applied to). In order to avoid namespace pollution, put the global variable in a namespace of your own. The example below uses a global variable in a made-up namespace (`scrap`) to compute partial sums for the numerical variable `x.weight`.

  ```
  $scrap::partial_sum += %x.weight%;
  ```

  Of course, this expression will only work once. After that, the variable `$scrap::partial_sum` must be reset to zero. As long as the first row in the data set has an `id` value of 1, we can use the following trick (be careful when using the **UCS::DS::Memory** module, where index activation might change the order of the rows).

  ```
  $scrap::partial_sum = 0 if %id% == 1;
  $scrap::partial_sum += %x.weight%;
  ```

COPYRIGHT

1.4 ucsam

Association measures in UCS/Perl

INTRODUCTION

The statistical analysis of cooccurrence data is usually based on **association measures**, mathematical formulae that compute an **association score** from the joint and marginal frequencies of a pair type (which are called a **frequency signature** in UCS. This score is a single floating-point number indicating the amount of statistical association between the components of the pair type. Association measures can often be written conveniently in terms of a **contingency table** of observed frequencies the corresponding expected frequencies under the null hypothesis that there is no association.

For instance, the word pair *black box* occurs 123 times in the *British National Corpus* (BNC), so its joint frequency is *f = 123*. The adjective *black* has a total of 13,168 occurrences, and the noun *box* has 1,810 occurrences, giving marginal frequencies of *f1 = 13,168* and *f2 = 1,810*. From these data, the **MI** measure computes an association score of *1.4*, while the **log.likelihood** measure computes a score of *567.72*. Both scores indicate a clear positive association, but they cannot be compared directly: each measure has its own scale.

A more detailed explanation of contingency tables and association scores as well as a comprehensive inventory of association measures with equations given in terms of observed and expected frequencies can be found on-line at *http://www.collocations.de/AM/*. Also see the ucsfile manpage to find out how frequency signatures, contingency tables and association scores are represented in UCS **data set** files.

UCS/Perl supports more than 40 different association measures and variants. In order to keep them managable, the measures are organised in several **packages**: a core set of widely-used "standard" measures is complemented by add-on packages for advanced users. Each package is implemented by a separate Perl module. Consult the module's manpage for a full listing of measures in the package and detailed descriptions. Listings of add-on packages, association measures, and some additional information can also be printed with the **ucs-list-am** program (see the ucs-list-am manpage).

Currently, there are two add-on packages in addition to the standard measures.

UCS::AM (the "standard" measures)

This core set contains all well-known association measures such as **MI**, **t-score**, and **log-likelihood** (see the listing in the Section §1.4 below). These measures are also made available by various other tools (e.g. the NSP toolkit, see *http://www.d.umn.edu/~tpederse/nsp.html*) and they have often been used in applications as well as for scientific research. The **UCS::AM** package also includes several other "simple" measures that are inexpensive to compute and numerically unproblematic.

Association measures in the core set can be thought of as the "built-in" measures of UCS/Perl (although the add-on packages are also part of the distribution). They are automatically supported by tools such as **ucs-add**, while the other packages have to be loaded explicitly (see below).

See the UCS::AM manpage for details.

UCS::AM::HTest (measures based on hypothesis tests)

Many association measures are based on asymptotic statistical hypothesis tests. The test statistic is used as an association score and can be interpreted (i.e. translated into a **p-value**) with the help of its known limiting distribution. The **UCS::AM::HTest** package provides p-values for all such association measures as well as the "original" two-tailed versions of some tests (the core set includes only one-tailed versions).

See the UCS::AM::HTest manpage for details.

UCS::AM::Parametric (parametric measures)

A new approach where the equation of a parametric association measure is not completely fixed in advance. One or more parameters can be adjusted to obtain a version of the measure that is optimised for a particular task or data set. Control over the parameters is only available through the programming interface. For command-line use, special versions of these measures are provided with a pre-set parameter value, which is indicated by the name of the measure.

See the UCS::AM::Parametric manpage for details.

In UCS/Perl scripts both the standard measures and the add-on packages have to be loaded with **use** statements (e.g. `use UCS::AM;` for the core set). Association measures are implemented as **UCS::Expression** objects (see the UCS::Expression manpage). The **UCS** module maintains a registry of loaded measures with additional information and an evaluation function (see Section "ASSOCIATION MEASURE REGISTRY" in the UCS manpage). When one of the packages above is loaded, its measures are automatically added to this registry. Association scores can be computed more efficiently for in-memory data sets, using the **add** method in the **UCS::DS::Memory** module (see the UCS::DS::Memory manpage).

In the **ucs-add** program, the standard measures are pre-defined, and extension packages can be loaded with the `-x` option. Only the last part of the package name has to be specified here (e.g. `HTest` for the **UCS::AM::HTest** package). It is case-insensitive and may be abbreviated to a unique prefix (so both `-x htest` and `-x ht` work as well). See the ucs-add manpage for more information on how to compute association scores with the **ucs-add** program.

SOME ASSOCIATION MEASURES

This section briefly lists the most well-known association measures available in UCS/Perl, all of which are defined in the "standard" package **UCS::AM**. See the on-line resource at *http://www.collocations.de/AM/* for fully equations and the UCS::AM manpage for details.

MI (Mutual Information)

The mutual information (MI) measure is a maximum-likelihood for the (logarithmic) *strength of the statistical association* between the components of a pair type. It was introduced into the field of computational lexicography by Church & Hanks (1990), who derived it from the information-theoretic notion of *point-wise mutual information*. Positive values indicate positive association while negative values indicate dissociation (where the components have a tendency *not* to occur together).

Note that unlike the original version of Church & Hanks (1990), the UCS implementation computes a base 10 logarithm.

t.score (t-score)

The MI measure is prone to overestimate association strength, especially for low-frequency cooccurrences. Church *et al.* (1991) use a version of Student's t test (whose test statistics is called a *t-score*) to ensure that the association detected by MI is supported by a *significant* amount of evidence. Although their application of Student's test is highly questionable, the combination of MI and t.score has become a *de facto* standard in British computational lexicography.

chi.squared, chi.squared.corr (chi-squared test)

Pearson's chi-squared test is the standard test for statistical independence in a *2 x 2* contingency table, and is much more appropriate as a measure of the *significance of association* than t.score. Despite its central role in mathematical statistics, it has not been very widely used on cooccurrence data. In particular, t.score was found to be much more useful for the extraction of collocations from text corpora (cf. Evert & Krenn, 2001).

The "textbook" form of Pearson's chi-squared test is a two-tailed version that does not distinguish between positive and negative association. The chi.squared measure implemented in UCS/Perl has been converted to a one-sided test with the help of a heuristic decision rule. Since contingency tables often contain cells with small values, Yates' continuity correction should be applied to the test statistic (chi.squared.corr).

log.likelihood (likelihood ratio test)

Dunning (1993) showed that the disappointing performance of chi.squared in collocation extraction tasks is due to a drastic overestimation of the significance of low-frequency cooccurrences (because of a approximation to its limiting distribution). He suggested to use a likelihood ratio test instead, whose natural logarithm has the same limiting distribution as chi.squared. Under the name *log-likelihood*, this association measure has become a generally accepted standard in the field of computational linguistics.

Like the chi-squared test, the likelihood ratio test is two-sided, and the log.likelihood measure has been converted to a one-sided test with the same heuristic decision rule. Both chi.squared and log.likelihood return the value of their test statistic, which has to be interpreted in terms of the known limiting distribution. More meaningful **p-values** for both measures are available in the UCS::AM::HTest package.

Fisher.pv (Fisher's exact test)

Although log.likelihood achieves a much better approximation to its limiting distribution than chi.squared (or chi.squared.corr), it is still an asymptotic and provides only an approximate p-value. Pedersen (1996) argued in favour of Fisher's exact test for the independence of rows and columns in a contingency table, in order to remove the remaining inaccuracy of the log-likelihood ratio. A drawback of Fisher's test is that it is numerically expensive and that naive implementations can easily become unstable.

The Fisher.pv measure implements a one-sided test. It returns an exact **p-value**, which can be compared directly with the p-values of chi.squared and log.likelihood.

Dice (Dice coefficient)

The Dice coefficient is a measure from the field of information retrieval, which has been used by Smadja (1993) and others for collocation extraction. Like MI, it is a maximum-likelihood estimate of *association strength*, but its definition of "strength" differs greatly from pointwise mutual information. It suffers from the same overestimation problem as MI, which is mitigated by its different approach to association strength, though.

References Church, K. W. and Hanks, P. (1990). Word association norms, mutual information, and lexicography. *Computational Linguistics* **16**(1), 22-29.

Church, K. W.; Gale, W.; Hanks, P.; Hindle, D. (1991). Using statistics in lexical analysis. In: *Lexical Acquisition: Using On-line Resources to Build a Lexicon*, Lawrence Erlbaum, pages 115-164.

Dunning, T. (1993). Accurate methods for the statistics of surprise and coincidence. *Computational Linguistics* **19**(1), 61-74.

Evert, S. and Krenn, B. (2001). Methods for the qualitative evaluation of lexical association measures. In: *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics*, Toulouse, France, pages 188-195.

Pedersen, T. (1996). Fishing for exactness. In: *Proceedings of the South-Central SAS Users Group Conference*, Austin, TX.

Smadja, F. (1993). Retrieving collocations from text: Xtract. *Computational Linguistics* **19**(1), 143-177.

UCS CONVENTIONS

UCS/Perl uses some conventions for the names of association measures and the computed association scores, which are described in this section. It is important to be aware of such conventions, especially when they deviate from those used by other software packages.

The **names of association measures** are taken from the on-line inventory at *http://www.collocations.de/AM/*. Hyphen characters (`-`) are replaced by periods (`.`) to conform with the UCS standards (see the ucsfile manpage). Capitalisation is preserved (`MI` and `Fisher.pv`, but `log.likelihood`) and subscripts are included in the name, separated by a period (`chi.squared.corr`, where `corr` is a subscript in the original name).

Association scores are always arranged so that **higher scores** indicate stronger (positive) association, applying a transformation to the original values if necessary. In the one-sided versions of two-sided tests (e.g. `chi.squared` and `log.likelihood`), negative scores indicate negative association (while positive scores indicate positive association). Scores close to zero are a sign of statistical independence. Some other measures such as `MI` also have this property, but many do not (e.g. `Fisher.pv` or `Dice`).

"Explicit" logarithms in the equation of an association measure are usually taken to the **base 10** (e.g. in the `MI` measure). This is not the case when the association score is not interpreted as a logarithm (e.g. the `log.likelihoood`, which is a test statistic approximating a known limiting distribution) and the natural logarithm is required for correct interpretation. The use of base 10 logarithms is always pointed out in the documentation (see the UCS::AM manpage). The logarithm of infinity if represented by a large floating-point value returned by the **inf** function (from the UCS::Expression::Func module). Comparison with `+inf()` and `-inf()` can be used to detect a positive or negative infinite value.

The scores of association measures with the extension `.pv` represent a p-value (from an exact test or the approximate p-value of an asymptotic test). Unlike most other scores, p-values can be compared directly between different measures. They are represented as **negative base 10 logarithms**, so the association score 3.0 corresponds to a p-value of 0.001 = 1e-3 (`+inf()` stands for zero probability, usually the result of an underflow error).

COPYRIGHT

2 UCS/Perl Programs

2.1 ucsdoc

UCS front-end to perldoc

SYNOPSIS

```
ucsdoc [-tk|-ps|-t] [options] PageName | ModuleName | ProgramName
```

DESCRIPTION

ucsdoc is a front-end to the **perldoc** program, which sets the required library paths for the UCS/Perl manpages. Standard Perl documentation is available through **ucsdoc** as well.

With the `-t` option, the manpage is formatted in plain ASCII, without highlighting.

With the `-ps` option, the manpage is formatted in PostScript for printing. The PostScript code is displayed on stdout so that it can be re-directed into a file or piped into a print command.

With the `-tk` option, the manpage is displayed in a Perl/Tk window, provided that the **Tk** and **Tk::Pod** modules are installed.

Only one of the three formatting options may be specified.

All other command-line arguments are passed to the **perldoc** program. Type `perldoc -h` and `perldoc perldoc` for more information on the available options.

COPYRIGHT

2.2 ucs-config

Automatic configuration of UCS/Perl scripts

SYNOPSIS

```
ucs-config

ucs-config [--version | --base-dir | --perl-dir | --bin-dir | --lib-dir | --R-bin]
ucs-config [-v | --base | --perl | --bin | --lib | -R]

ucs-config ucs-script.pl ucs-script.R ...

ucs-config --run [options] one-liner.perl
ucs-config --run [options] -e '...'
ucs-config -e '...'
```

DESCRIPTION

The **ucs-config** program is used to print information about the installed UCS/Perl version and directories, as well as for the automatic configuration of UCS/Perl scripts. The program can be run in four different modes.

Invoking **ucs-config** without any arguments prints the UCS splash screen and a configuration summary.

In the second mode, the program prints one item of configuration information selected with one of the following flags. This mode is most suitable for use in shell scripts and makefiles. Note that you are not allowed to specify more than one flag at a time.

```
--version    UCS version
--base-dir   root directory of the UCS system
--perl-dir   root directory of the UCS/Perl subsystem
--bin-dir    bin/ directory of UCS/Perl (contains UCS programs)
--lib-dir    lib/ directory of UCS/Perl (contains UCS modules)
--R-bin      fully qualified filename of the R interpreter
```

The third mode is used to in-place edit Perl and R scripts so that they can load the **UCS** modules and libraries. For **Perl scripts**, **ucs-config** inserts a suitable shebang (`#!`) line, invoking the Perl interpreter for which UCS is configured together with the necessary include paths. For **R scripts** (which are recognised by their extension `.R` or `.S`), **ucs-config** looks for a line containing the command `source("../ucs.R")` in the script, and inserts the correct path there. Please make sure that this line does not contain any other commands.

The final mode, introduced by the command-line switch `--run`, invokes the Perl interpreter with the correct UCS library path and (almost) all **UCS modules** pre-loaded (including the standard association measures from **UCS::AM**, but none of the add-on packages). The remaining command-line arguments are passed through to the Perl interpreter, which is *really cool* for writing one-liners in **UCS/Perl**. The flag `-e` is an abbreviation of `--run -e`, but does not allow any options to be passed to the interpreter.

COPYRIGHT

2.3 ucs-tool

Execute UCS/Perl scripts from contrib/ tree

SYNOPSIS

```
ucs-tool --list [--category | --category=<cat>]
ucs-tool --doc <tool> [<ucsdoc options>]
ucs-tool [--category=<cat>] <tool> ...
```

DESCRIPTION

In addition to the UCS/Perl programs, which perform general tasks and will be of interest to most users, the UCS distribution includes a number of UCS/Perl scripts for more specific applications. These scripts are not directly accessible as command-line programs. They are organised into a hierarchical set of categories in the *contrib/* directory tree, and can be invoked through the **ucs-tool** program. If you want to add your own scripts to this tree, read the section on §2.3 below.

LISTING CONTRIBUTED SCRIPTS When the `--list` (or `-l`) option is specified, **ucs-tool** lists all available UCS/Perl scripts from the *contrib/* tree, grouped by category. Add the option `--category` (or `--cat` or `-c`) for a listing of category names and descriptions (without the individual tools). You can also use the special short form `ucs-tool -lc` for this purpose. When an argument is given for `--category`, only scripts from the specified category are listed (the category name is case-insensitive).

Some scripts may provide manual pages in the form of embedded POD documentation. Such manual pages can be displayed with the `--doc` (or `-d`) flag, followed by the name of the script. See the section on §2.3 below for details on how script names are matched. **ucs-tool** uses the **ucsdoc** program to format manual pages and accepts **ucsdoc** options (such as `-ps` and `-tk`) *after* the tool name.

SCRIPT INVOCATION In order to invoke one of the contributed UCS/Perl scripts, simply specify its name (as shown by the `--list` option), followed by command-line arguments for the selected script, e.g.

```
ucs-tool dispersion-test -m 3 -N 100000 -k 100 -V 2500
```

All contributed scripts should include a short help page that can be displayed with the `--help` (or `-h`) option. Note that this is a script option and therefore must be specified *after* the script name:

```
ucs-tool dispersion-test --help
```

Recall that full manual pages, when available, can be displayed with the `--doc` option specified *before* the script name (as described above).

Script names are case-insensitive, and it is sufficient to specify a unique prefix of the name. For instance, you can invoke the **print-documentation** script with the short name `ucs-tool print` or `ucs-tool print-doc`. It may be easier to find a unique prefix when the search space is reduced to a specific category with the `--category` (or `-c`) option.

WRITING CONTRIBUTED SCRIPTS

Contributed UCS/Perl scripts are collected in a directory tree rooted in *System/Perl/contrib/*. Each subdirectory corresponds to a script category. These categories are organised hierarchically according to the directory structure (for instance, `--list --category=Import` lists all scripts found in the directory *Import/* and its subdirectories, such as *Import/NSP/* and *Import/CWB/*). The file *CATEGORIES* contains a listing of all known categories with short descriptions (category names and descriptions must be separated by a single TAB character).

If you want to add your own UCS/Perl scripts to the repository, you should put them in the *Local/* directory (which is reserved for scripts that are not part of the UCS distribution). This is often the easiest way to make a UCS/Perl script available to all users of a UCS installation. Note that script files *must* have the extension `.perl` or `.pl`, which is not part of the script name (e.g., the script **nsp2ucs** in the category **Import/NSP** corresponds to the disk file *Import/NSP/nsp2ucs.perl* in the *contrib/* tree). You can also put your script in a different category or define your own categories (which you must add to the *CATEGORIES* file), but this will interfere with upgrading to a new UCS release. You are encouraged to share scripts with other users. To do so, please send them to the author (or maintainer) of the UCS system, indicating which category they should be included in.

Unlike ordinary UCS/Perl scripts, scripts placed in the *contrib/* tree do not have to be configured with **ucs-config**. They also do not have to be executable and start with a shebang (`#!`) line. When invoked with the **ucs-tool** program, the necessary settings are made automatically. Contributed scripts that require "private" modules (which are not installed in a public directory) can place them in a subdirectory named *lib/* (relative to the location of the script file), or in further subdirectories as required by the module's name. The *lib/* directory tree is automatically added to Perl's search path. Necessary data files should be wrapped in Perl modules and stored in the *lib/* subtree as well. For instance, assume that a script named **my-script** in the **Local** category (corresponding to the script file *Local/my-script.perl*) uses the private module **My::Functions**. This module can automatically be loaded (with `use My::Functions;`) from the file *Local/lib/My/Functions.pm* in the *contrib/* directory tree.

All contributed UCS/Perl scripts should include a short help page describing the script's function and command-line arguments, which is displayed when the script is invoked with `--help` or `-h`. Script authors are also encouraged to write full manual pages as embedded POD documentation (which can then be displayed with `ucs-tool --doc`), but these are not mandatory.

COPYRIGHT

2.4 ucs-list-am

List built-in association measures and add-on packages

SYNOPSIS

```
ucs-list-am [-v | -c | -t | -f <f,f1,f2,N>]
            [-x <package> | -p <package>] [<am1> <am2> ...]

ucs-list-am --list
```

DESCRIPTION

This program is a convenient front-end to the registry of association measures maintained by the **UCS** module. It can be used to print a list of built-in association measures, add-on packages, and display additional information about the measures (where available). Detailed information about the measures can be found in the UCS::AM manpage and the respective manpages of the extension packages. See the ucsam manpage for an introduction and overview.

```
ucs-list-am --list
```

With the `--list` (or `-l`) option, **ucs-list-am** lists all available add-on packages.

```
ucs-list-am [<options>] [<am1>, <am2>, ...]
```

When **ucs-am-list** is called without arguments, it prints the names of all built-in association measures on stdout, each one followed by a short one-line description of the measure. Specific association measures can be selected by giving their names as command-line arguments. UCS wildcard patterns (see the ucsexp manpage) will list all matching measures.

The `--extra` (or `-x`) option can be used to load one or more add-on packages so that the association measures from these packages will be included in the listing (in addition to the built-in measures). Its argument is a comma-separated list of package names, which are case-insensitive and may be abbreviated to unique prefixes. For instance, both `--extra=HTest,Parameteric` an `-x htest,param` will load the **UCS::AM::HTest** and **UCS::AM::Parametric** packages. The special keyword `ALL` loads all available AM packages.

The `--package` (or `-p`) option is used to list the association measures from a single package (*without* the built-in measures). Again, the package name is case-insensitive and may be abbreviated to a unique prefix. Note that the `--package` option cannot be used to load multiple packages.

The amount of information provided can be controlled with the `--verbose` (or `-v`), `--code` (or `-c`), and `--terse` (or `-t`) options. In `--terse` mode, only the names of packages are printed, so that the output can be easily processed by other programs. In `--verbose` mode, the name of each association measure is immediately followed by a one-line description (in parentheses). When available, one or more lines of additional comments will also be shown. In `--code` mode, the output consists of the name of each measure, followed by its implementation (as a UCS expression), followed by a blank line. For parameteric measures, a list of parameters and their default values is shown on a separate line between the name and the implementation.

Alternatively, a frequency signature can be specified as an argument to the `--frequencies` (or `-f`) option. The expected format is a comma-separated list of four integers, representing the variables `f`, `f1`, `f2` and `N`. In this case, association scores for all selected measures are computed on the specified frequency signature. Note that it is not possible to compute scores for different frequency signatures with a single invocation of the **ucs-list-am** tool.

COPYRIGHT

2.5 ucs-make-tables

Compute contingency tables from a sequence of pair tokens

SYNOPSIS

```
... | ucs-make-tables [-v] [--sort | -s] [--sample-size=<n> | -N <n>]
                      [--threshold=<n> | -f <n>]  data.ds.gz

... | ucs-make-tables [-v] [-s] [-N <n>] [-f <n>]
                      [--dispersion [--chunk-size=<n>] ]  data.ds.gz

... | ucs-make-tables [-v] [-s] [-N <n>] [-f <n>] --segments data.ds.gz
```

DESCRIPTION

This utility computes frequency signatures and constructs a UCS data set for a stream of pair tokens (or segment-based cooccurrence data) read from STDIN. It is usually applied to the output of a cooccurrence extraction tool in a command-line pipe. The input can also be read from a file (with a < redirection), or decompressed on the fly with (`gzip -cd` or `bzip2 -cd`). The resulting data set is written to the file specified as the single mandatory argument on the command-line.

ucs-make-tables operates in two different modes for **relational** and **positional** (segment-based) cooccurrences. These two modes are described separately in the following subsections. They take the same command-line options and arguments, as described in the section §2.5 below. Distance-based positional cooccurrences are not supported, as they usually require direct access to the source corpus in order to determine the precise window size.

Relational Cooccurrences By default, **ucs-make-tables** operates in a mode for relational cooccurrences. In this mode, the input line format is

```
<l1> TAB <l2>
```

Each such line represents a pair token with labels <l1> and <l2> (i.e. a pair token that belongs to the pair type *(l1,l2)*). For dispersion counts (see below), the input lines should preserve the order in which the corresponding pair tokens appear in the corpus. When dispersion is measured with respect to pre-annotated parts (e.g. paragraphs or documents) rather than equally-sized parts, the input must contain an extra column with unique part identifiers:

```
<l1> TAB <l2> TAB <part_id>
```

Note that all pair tokens from a given part must form an uninterrupted sequence in the input, otherwise the dispersion counts will not be correct.

Segment-based Cooccurrences The mode for segment-based cooccurrences is activated with the `--segments` (or `-S`) option. In this mode, each segment is represented by a sequence of four lines in the input stream, called a **record**:

1. <segment_id> [TAB <part_id>]
2. The labels of all tokens in the segment that can become *first* components of pairs, separated by TABs.
3. The labels of all tokens in the segment that can become *second* components of pairs, separated by TABs.
4. A blank separator line.

Duplicate strings on the second or third line will automatically be ignored. The <segement_id> on the first line is currently ignored. The optional <part_id> can be used to compute dispersion counts for pre-annotated parts. All segments that belong to a given part must appear in consecutive records, otherwise the dispersion counts will not be correct.

A prototypical example of the segment-based approach are lemmatised noun-verb cooccurrences within sentences. In this case, each record in the input stream corresponds to a sentence. The first line contains an unimportant sentence identifier. The second line contains the lemma forms of all nouns in the sentence (note that duplicates are automatically removed), and the third line contains the lemma forms of all verbs in the sentence. In order to compute the dispersion of cooccurrences across documents (i.e. *document frequencies* in the terminology of information retrieval), unique document identifiers have to be added to the first line.

COMMAND LINE

The general form of the **ucs-make-tables** command is

```
... | ucs-make-tables [--verbose | -v] [--sort | -s]
                      [--threshold=<t> | -f <t>]
                      [--sample-size=<n> | -N <n>]
                      [--dispersion [--chunk-size=<s>]]
                      [--segments]
                      data.ds.gz
```

With the `--verbose` (or `-v`) option, some progress information (including the number of pair tokens or segments, as well as the number of pair types encountered so far) is displayed while the program is running. When `--sort` (or `-s`) is specified, the resulting data set is sorted in ascending alphabetical order (on `l1` first, then `l2`). Of course, the data set file can always be re-sorted with the **ucs-sort** utility. When a frequency threshold <t> is specified with the `--threshold` (or `-f`) option, only pair types with cooccurrence frequency `f >=` <t> will be saved to the data set file (but they are still included in the marginal frequency counts of relational cooccurrences, of course). This option helps keep the size of data sets extracted from large corpora manageable.

When `--sample-size` (or `-N`) is specified, only the first <n> pair tokens (or segment records) read from STDIN will be used, so that the sample size `N` of the resulting data set is equal to <n>. This option is mainly useful when computing dispersion counts on equally-sized parts (see below), but it has some other applications as well.

With the `--dispersion` (or `-d`) option, dispersion counts are added to the data set and can then be used to test the random sample assumption with a **dispersion test** (see Baayen 2001, Sec. 5.1.1). In order to do so, the token stream is divided into equally-sized **parts**, each one containing the number <s> of pair tokens specified with the `--chunk-size` (or `-c`) option. For segment-based cooccurrences, each part will contain cooccurrences from <s> segments. When the total number of pair tokens (or segments) is not an integer multiple of <s>, a warning message will be issued. In this case, it is recommended to adjust the number of tokens with the `--sample-size` option described above.

The dispersion count for each pair type, i.e. the number of parts in which it occurs, is stored in a variable named `n.disp` in the resulting data set file. In addition, the number of parts and the part size are recorded in the global variables `chunks` and `chunk.size`. When the part size is not specified, dispersion counts can be computed for pre-annotated parts, which must be identified in the input stream (see above). In this case, `chunk.size` is not defined as the individual parts may have different sizes. **NB:** The use of pre-annotated parts is discouraged, since the mathematics of the dispersion test assume equally-sized parts.

EXAMPLES

If you have installed the IMS Corpus Workbench (CWB) as well as the CWB/Perl interface, you can easily extract relational adjective+noun cooccurrences from part-of-speech tagged CWB corpora. The **ucs-adj-n-from-cwb.perl** script supplied with the UCS system supports several

tagsets for German and English corpora. It can easily be extended to other tagsets, languages, and types of cooccurrences (as long as they can be identified with the help of part-of-speech patterns).

The following example extracts adjective+noun pairs with cooccurrence frequency `f >= 3` from the CWB demonstration corpus `DICKENS` (ca. 3.4 million words), and saves them into the data set file `dickens.adj-n.ds.gz`. The shell variable `$UCS` refers to the *System/* directory of the UCS installation (as in the UCS/Perl tutorial).

```
$UCS/Perl/tools/ucs-adj-n-from-cwb.perl  penn  DICKENS
     |  ucs-make-tables  --verbose --sort --threshold=3  dickens.adj-n.ds.gz
```

(Note that the command must be entered as a single line in the shell.)

Extraction from the `DICKENS` corpus produces approximately 122990 pair tokens. In order to apply a dispersion test with a chunk size of 1000 tokens each, the sample size has to be limited to an integer multiple of 1000:

```
$UCS/Perl/tools/ucs-adj-n-from-cwb.perl  penn  DICKENS
     |  ucs-make-tables  --verbose --sort --threshold=3 --sample-size=122000
                        --dispersion --chunk-size=1000  dickens.disp.ds.gz
```

A dispersion test for pair types with `f <= 5` can then be performed with the following command, showing a significant amount of underdispersion at all levels.

```
$UCS/Perl/tools/ucs-dispersion-test.perl -v -m 5 dickens.disp.ds.gz
```

Segment-based data can be obtained from a CWB corpus with the **ucs-segment-from-cwb.perl** script. The following example extracts nouns and verbs cooccurring within sentences. A frequency threshold of 5 is applied in order to keep the amount of data (and hence the memory consumption of the **ucs-make-tables** program) manageable.

```
$UCS/Perl/tools/ucs-segment-from-cwb.perl -f 5 -t1 "VB.*" -t2 "NN.*" DICKENS s
     | ucs-make-tables --verbose --segments --threshold=5 dickens.n-v.ds.gz
```

REFERENCES

Baayen, R. Harald (2001). *Word Frequency Distributions.* Kluwer, Dordrecht.

IMS Corpus Workbench (CWB): *http://www.ims.uni-stuttgart.de/projekte/CorpusWorkbench/*

COPYRIGHT

2.6 ucs-summarize

Compute statistical summaries for variables in UCS data set

SYNOPSIS

```
ucs-summarize [-v] [-m] f f1 f2 FROM data.ds.gz

ucs-summarize [-v] [-m] am.%.pv FROM data.ds.gz

ucs-summarize [-v] [-m] data.ds.gz
```

DESCRIPTION

This program computes short statistical summaries of numerical variables in a UCS data set. The general form of the **ucs-summarize** command is

```
ucs-summarize [-v] [-m] <variables> FROM <input.ds>
```

where <`variables`> is a whitespace-separated list of variable names or wildcard expression, and the data set is read from the file specified as <`input.ds`>. Wildcard expressions may need to be quoted to avoid interpretation by the shell. When the list of variables is omitted (including the keyword `FROM`), summaries are generated for all variables in the data set. In verbose mode (`--verbose` or `-v` option), some progress information is shown while computing the summary.

So far, the statistical summary includes the **minimum** (`min.`), **maximum** (`max.`), **mean** (`mean`), **empirical variance** (`var.`), and the **empirical standard deviation** (`s.d.`). In addition, the number of missing values (`NA's`) is reported.

When `--memory` (or `-m`) is specified, the data set will be read into memory first. In addition to the ordinary statistical summary, the **absolute minimum** (`abs.min.`, the smallest non-zero absolute value), **absolute maximum** (`abs.max.`), and **granularity** (`gran.`, smallest difference between any two unequal values) are computed in this mode.

COPYRIGHT

2.7 ucs-select

Select rows and/or columns from UCS data set

SYNOPSIS

```
ucs-select --count FROM data.ds.gz WHERE '%011% < %E11%'

ucs-select '*' 'am.%.pv' FROM data.ds.gz INTO new.ds.gz

ucs-select '%' FROM data.ds.gz WHERE 'not defined %b.accept%'
```

DESCRIPTION

This program is used to select rows and/or columns from a UCS data set file, very much like a `SELECT` statement in SQL. The general form of the **ucs-select** command is

```
ucs-select [--verbose | -v] (<variables> | --count)
           [ FROM <input.ds> ] [ WHERE <condition> ] [ INTO <output.ds> ]
```

<`variables`> is a whitespace-separated list of variable names or wildcard patterns (see the ucsexp manpage), which are matched against the columns of the data set file <`input.ds`>. The list of variables may not be omitted: use `'%'` to select *all* columns, and `--count` to display the number of matching rows only. Note that wildcard patterns may need to be quoted individually (because they contain shell metacharacters).

<`condition`> is a UCS expression (see the ucsexp manpage) used to select rows from the data set for which it evaluates to a true value. When the `WHERE` clause is omitted, all rows are selected. Note that <`condition`> must be a single argument and will usually have to be quoted (single quotes are highly recommended).

The input data set file <`input.ds`> defaults to STDIN (when omitted). The resulting table is printed on STDOUT in UCS data set file format (see the ucsfile manpage), and can be written to a data set file <`output.ds`> with the optional `INTO` clause.

With the `--verbose` (or `-v`) option, some progress information is displayed while the program is running.

COPYRIGHT

2.8 ucs-add

Add variables (association scores) to UCS data set

SYNOPSIS

```
ucs-add [-v] [-m] am.t.score am.Fisher.pv TO data.ds.gz INTO new.ds.gz

ucs-add [-v] [-m] -x HTest am.%.pv TO data.ds.gz INTO new.ds.gz

ucs-add [-r] r.% TO data.ds.gz INTO new.ds.gz
```

DESCRIPTION

This program is used to add variables (**association scores**, **rankings**, **derived variables**, or arbitrary **UCS expressions** entered on the command line) to a UCS data set. If a variable is already defined in the data set, its values will be overwritten.

The general form of the **ucs-add** command is

```
ucs-add [--verbose | -v] [--memory | -m] [--extra=<list> | -x <list>]
        <variables> [ TO <input.ds> ] [ INTO <output.ds> ]
```

where <variables> is a whitespace-separated list of variable specifications (see the section on §2.8 below for details). An additional `--randomize` option is only useful when adding rankings:

```
ucs-add [--verbose | -v] [--extra=<list> | -x <list>] [--randomize | -r]
        <variables> [ TO <input.ds> ] [ INTO <output.ds> ]
```

The data are read from the file <input.ds>, and the resulting data set with the new annotations is written to the file <output.ds>. When they are not specified, the input and output files default to STDIN and STDOUT, respectively.

Variable specifications and file names may need to be quoted individually (when they contain shell metacharacters or whitespace).

Normally, the **ucs-add** program processes the data set one row at a time, so that <input.ds> and <output.ds> must not refer to the same file. When `--memory` (or `-m`) is specified, the entire data set is read into memory, annotated, and then written back to the output file. In this case, <input.ds> and <output.ds> may be identical. This mode is automatically activated when any rankings are added to the data set.

In both modes of operation, variables are added in the order in which they are given on the command-line, so variable specifications (rankings and user-defined expressions) may refer to any of the previously introduced variables.

With the `--verbose` (or `-v`) option, some debugging and progress information is displayed while the program is running. The `--extra` (or `-x`) option loads additional built-in association measures (see the section on adding §2.8 below for details).

VARIABLE SPECIFICATIONS

Association Scores Variables representing association scores are selected by specifying their variable names (which start with the prefix `am.`). The names may be given as UCS wildcard patterns (see the ucsexp manpage), which will be matched against the list of all supported association measures. Examples of useful wildcard patterns are `am.%` (all measures), `am.%.pv` (all measures that compute probability values), and `am.chi.squared.%` (all variants of Pearson's chi-squared test).

By default, only the basic association measures defined in UCS::AM are supported. Other AM packages (see the UCS::AM manpage for a list of add-on packages) can be loaded with the `--extra` (or `-x`) option. The argument is a comma-separated list of package names (e.g. `--extra=HTest,Parametric` to load **UCS::AM::HTest** and **UCS::AM::Parametric**), which are case-insensitive and may be abbreviated to unique prefixes (so `-x htest,par` works just as well). Use `-x ALL` to load all available AM packages.

Rankings Variables representing association score rankings are selected by specifying their variable names (which start with the prefix `r.`). In order to compute a ranking, say `r.something`, the corresponding association scores (`am.something`) must be annotated in the data set. UCS wildcard patterns are matched against all association scores in the data set (but not against other built-in association measures). Rankings can also be computed for user-defined measures, provided that their association scores are annotated. In order to compute a ranking for a built-in association measure that is not available in the data set, both the association score and the ranking variable must be specified. The example

```
ucs-add -m am.% r.% TO data.ds.gz INTO data.ds.gz
```

adds associations scores and rankings for the basic built-in association measures to the data set `data.ds.gz`.

Ties are not resolved in the rankings, so pair types with identical association scores share the same rank. The rank assigned to such a group of pair types is the lowest free rank (as in the Olympic Games) rather than the average of all ranks in the group (as is often done in statistics). With the `--random` (or `-r`) option, ties are resolved in a random fashion. When association scores for the `random` measure are pre-annotated (i.e. the `am.random` variable is present in the data set), these are used for the randomization so that the ranking is reproducible.

Derived Variables Any variable names or wildcard patterns that do not match one of the built-in association measures are matched against the list of derived variables, which can be computed automatically from the frequency signatures of pair types. See the ucsfile manpage for a complete list of derived variables. Examples of useful patterns are `E*` (expected frequencies), `lp*` (logarithmic coordinates), and `e b m` (*(e,b,m)*-coordinates).

User-Defined Expressions A user-defined variable specification is a UCS expression (see the ucsexp manpage) of the form

```
<var> := <expression>
```

where <`var`> is the name of a user-defined variable, association score, or ranking (without surrounding `%` characters). This variable is added to the input data set if necessary and set to the values computed by the UCS expression <`expression`>. The example below computes association scores for a compound measure `mixed` from the rankings according to two other measures (which must both be annotated in the data set).

```
am.mixed := -max(%r.t.score%, %r.dice%)
```

Note that it isn't possible to compute the corresponding ranking `r.mixed` directly.

COPYRIGHT

2.9 ucs-join

Join rows and variables from two UCS data sets

SYNOPSIS

```
ucs-join [--match-on var1,var2,...] data1.ds.gz data2.ds.gz

ucs-join [--add] [--update] [--multiple] [-m var1,var2,...]
         data1.ds.gz data2.ds.gz INTO new.ds.gz

ucs-join [--add] [--update] [--multiple] [-m var1,var2,...]
         data1.ds.gz WITH am.% FROM data2.ds.gz INTO new.ds.gz
```

DESCRIPTION

This program can be invoked in three different ways. The short form

```
ucs-join  [-v] [-m <var>,...]  <ds1>  <ds2>
```

compares two data sets <ds1> and <ds2>. In particular, the number of rows common to both data sets and the numbers of rows unique to either one of the data sets are reported. Rows are matched on the **pair types** they represent, i.e. the variables l1 and l2. Differences in the id value or any other annotations are ignored. The **coverage** is the proportion of pair types in <ds1> that are also contained in <ds2>.

With the --verbose (or -v) switch, some progress information is displayed while the program is running. The --match-on (or -m) flag specifies a comma-separated list of variables to use for matching rows (instead of l1 and l2). Note that the combination of their values must be unique for every row within each data set.

The second form

```
ucs-join  [-v] [--add] [--update] [--multiple] [-m <var>,...]
          <ds1> <ds2> INTO <ds3>
```

adds variables and/or rows from the data set <ds2> to <ds1>. Rows from the two data sets are matched on the l1 and l2 variables as above, unless this has been changed with the --match-on (or -m) flag. The combination of their values must uniquely identify rows in <ds2>, while duplicate rows in <ds1> are allowed in combination with the --multiple (or -M) option.

For matching rows, all variables from <ds2> are added to the annotations in <ds1>. Variables that are common to both data sets are overwritten with the values from <ds2> only when they are undefined (NA) in <ds1>, or when the --update (or -u) option has been given. For backward compatibility, the default setting can be explicitly selected with --no-overwrite (or -n). If --add or -a is specified, rows that appear only in <ds2> are added to <ds1> (with all variables that are not defined in <ds2> set to NA). The resulting data set is written to the file <ds3>.

The most general form

```
ucs-join  [-v] [--add] [--update] [--multiple] [-m <var>,...]
          <ds1> WITH <variables> FROM <ds2> INTO <ds3>
```

adds selected variables from <ds2> only. <variables> is a whitespace-separated list of variables names and wildcard patterns, which are matched against the variables of <ds2>. Variables can be renamed with specifiers of the form new.name=old.name (of course, wildcard patterns cannot be used here). The --add switch is rarely useful with this form of the **ucs-join** command.

ANNOTATION DATABASES

The **ucs-join** program is often used to add (manual) annotations from an **annotation database** file (`.adb`) to a data set, and to update annotation databases. For instance, the UCS distribution includes German PP+verb pairs extracted from the *Frankfurter Rundschau* corpus (*fr-pnv.ds.gz*) and an annotation database created by Brigitte Krenn (*pnv.adb.gz*). In order to check the **coverage** of the annotation database (i.e., how many of the pair types are already contained in the database), type

```
ucs-join -v fr-pnv.ds.gz pnv.adb.gz
```

This will show a coverage of 100%. Annotations from the database can now be added to the *fr-pnv.ds.gz* data set (the `--update` option is only relevant if *fr-pnv.ds.gz* is already annotated with the relevant variables):

```
ucs-join -v --update fr-pnv.ds.gz
         WITH 'b.*' FROM pnv.adb.gz INTO fr-pnv.annot.ds.gz
```

When an annotation database contains entries that have not been manually examined so far, these should be annotated with missing values (`NA`). The database can then be updated from a new file (in the same `.adb` format, say *new-pnv.adb*) with the following commands

```
mv pnv.adb.gz pnv.adb.BAK.gz
ucs-join -v pnv.adb.BAK.gz new-pnv.adb INTO pnv.adb.gz
```

If the file *new-pnv.adb* contains additional pair types (that haven't already been entered into the database), you should also specify the `--add` flag.

Recall that **ucs-join** will not overwrite existing annotations by default. If you want to correct mistakes in the annotation database, you need to specify the `--update` option in the command above. Note that missing values (`NA`) will *never* overwrite existing annotations in the first data set.

COPYRIGHT

2.10 ucs-sort

Sort UCS data set by one or more variables

SYNOPSIS

```
ucs-sort [-v] [-r] [data.ds.gz] BY am.t.score [INTO new.ds.gz]

ucs-sort [-v] [-r] [data.ds.gz] BY l2+ l1- ... [INTO new.ds.gz]
```

DESCRIPTION

This program sorts the rows of UCS data by one or more variables. The general form of the **ucs-sort** command is

```
ucs-sort [--verbose | -v] [--randomize | -r]
         [<input.ds>] BY <variables> [INTO <output.ds>]
```

where <`variables`> is a whitespace-separated list of variable names. A `+` or `-` character appended to a variable name selects ascending or descending order, respectively. The default order depends on the variable type (association scores are sorted in descending order).

The data set is read from STDIN by default, or from the file <`input.ds`> when it is specified. The sorted data set is printed on STDOUT, and can be saved into the file <`output.ds`> with the optional `INTO` clause.

When `--randomize` (or `-r`) is specified, ties are broken randomly, using the `am.random` measure if it is annotated in the data set. The `--verbose` (or `-v`) option displays some (minial) progress information.

EXAMPLES

The **ucs-sort** utility is often used in command-line pipes to sort data sets before viewing. Assuming that a data set file *candidates.ds.gz* is annotated with the necessary association scores, ranked candidate lists for the log-likelihood and t-score measures can be displayed with the following commands:

```
ucs-sort -r candidates.ds.gz BY am.log.likelihood | ucs-print -i
ucs-sort -r candidates.ds.gz BY am.t.score | ucs-print -i
```

ucs-sort can also be applied to the output of another UCS tool, e.g. **ucs-select**. The following command selects the 100 highest-ranked pair types from the data set file *candidates.ds.gz*, according to the log-likelihood measure, and displays them in alphabetical order, sorted by `l2` first. (Note that the command must be entered as a single line in the shell.)

```
ucs-add -v r.log.likelihood TO candidates.ds.gz
  | ucs-select -v '%' WHERE '%r.log.likelihood% <= 100'
  | ucs-sort BY l2 l1 | ucs-print -i
```

COPYRIGHT

2.11 ucs-info

Display information from header of UCS data set file

SYNOPSIS

```
ucs-info [-s [-v]] [-l] data.ds.gz
```

DESCRIPTION

This small utility displays information from the header of a data set file (comment lines and global variables).

With the `--size` (or `-s`) option, the actual size of the data set (i.e. the number of pair types) is also determined, which may be different from the size reported in the header. Note that this operation has to read the entire data set file and may take some time for larger data sets (use `--verbose` or `-v` to show progress information).

With the `--list` (or `-l`) option, the data set variables are listed together with their data types and optional comments.

COPYRIGHT

2.12 ucs-print

ASCII-format UCS data set for viewing and printing

SYNOPSIS

```
ucs-print [-i] [-p <lines>] [-d <digits>] data.ds.gz

ucs-print [-o <file>] [-ps [-2] [-l]] [-p <lines>] [-d <digits>] data.ds.gz

ucs-print [<options>] '*' 'am.%.pv' FROM data.ds.gz
```

DESCRIPTION

Format data set as ASCII table for inclusion in text files, on-line viewing (in a terminal window, with `--interactive` option), and printing (in PostScript format, with `--postscript` option). The **ucs-print** utility automatically adjusts column widths and chooses an appropriate format for floating-point numbers. Boolean attributes are displayed as `yes` and `no`, while missing values are shown as `NA`.

In the first forms of the command (used in the first two examples above), all variables are displayed (which usually results in a very wide table). The name of the data set may be omitted, in which case data is read from `STDIN`.

In the second form, variables can be selected with a whitespace-separated list of UCS wildcard patterns (see the ucsexp manpage) or by explicitly specifying the variable names. This feature can also be used to re-order the columns or display a variable in multiple columns. The `FROM` clause is mandatory in this mode, but data can be read from `STDIN` by using `-` as the name of the data set.

Note that there may be some delay while the data set is read into memory and analysed, especially without the `--pagesize` option.

OPTIONS

- `--help`, `-h`

  Prints short usage reminder.

- `--verbose`, `-v`

  Prints some (minimal) progress information on `STDERR`.

- `--output` *file*, `-o` *file*

  Write output to *file*, rather than printing it on `STDOUT`.

- `--postscript`, `-ps`

  Uses the **a2ps** program (see the a2ps(1) manpage) to create a PostScript version of the formatted table for printing. By default, the PostScript code will be shown on `STDOUT` (and *not* be sent to a printer). It can be saved into a file with the `--output` option. If the `--pagesize` option is used, each page will contain the specified number of rows and the table will be truncated if it is too wide. If this happens, try increasing the number of rows on the page or use `--landscape`. If the table still fails to fit, split the variables into two or more groups that are printed separately.

- `--landscape`, `-l`

  [In `--postscript` mode only.] Print pages in landscape orientation rather than portrait. Especially useful for wide tables.

- `--two-up`, `-2`

  [In `--postscript` mode only.] Print two pages on a single sheet, same as the `-2` option in **a2ps**. This option may give a more satisfactory result for very narrow tables (e.g. when showing only the pair types).

- `--interactive`, `-i`

  Send output to terminal pager (**less**) for interactive viewing. This option may not be used together with `--output`. The data will automatically be displayed in paged mode, with the page size adjusted to the height of the terminal window. If the screen size cannot be automatically determined, use the `--pagesize` option to activate paging explicitly. The page size should be set to the screen height (number of text lines) minus 4 for optimal results. Use `-p 0` to deactivate paging in interactive mode.

- `--pagesize` *n*, `-p` *n*

  Split data set into smaller tables of (up to) *n* rows each, which are separated by blank lines. Use of this option may improve the formatting quality, helps to avoid excessive columns widths, and reduces the delay before (partial) results can be displayed (especially for large data sets). By default, the entire data set is formatted as a single large table (unless `--interactive` was specified).

- `--digits` *n*, `-d` *n*

  Display floating-point numbers with a precision of approximately *n* significant digits. The actual number of digits shown may differ slightly when a fixed-point format is chosen by the formatter. The default is $n = 8$.

BUGS

The code used to determine the screen height in `--interactive` mode may not work on some platforms. It has only been tested under Linux so far. If you are using the **bash** shell, you might try `export LINES` before running the **ucs-print** tool.

COPYRIGHT

3 UCS/Perl Modules

3.1 UCS

Core library

SYNOPSIS

```
use UCS;

$UCS::Version;                                # UCS version
$UCS::Copyright;                              # UCS copyright string
$UCS::BaseDir;                                # base directory of UCS system
$UCS::PerlDir;                                # base directory of UCS/Perl

UCS::Die("Msg line 1", "Msg line 2", ...);    # really die (even in Tk loop)
UCS::Warn("Msg line 1", "Msg line 2", ...);   # warning message (may be caught by Tk)
UCS::Status("Message");                       # display status message in Tk window
UCS::Splash();                                # splash screen (may be shown during start-up)
$UCS::Verbose = 0;                            # suppress warnings
@unique_values = UCS::Unique(@list);          # remove duplicates from list

@vars = (@UCS::CoreVars, @UCS::DerivedVars);  # standard variable names (core and derived)
@matches = UCS::Match($pattern, @names);      # match variable names
$ok = UCS::ValidKey($key);                    # valid identifier, e.g as AM key
$ok = UCS::ValidName($name);                  # whether variable name is valid
$type = UCS::VarType($name);                  # "BOOL", "INT", "DOUBLE", "STRING"
($spec, $key) = UCS::SplitName($name);        # split am.*, r.*, or user-defined variable name

@registered_AMs = UCS::AM_Keys();             # keys for built-in AMs (when loaded)
if (UCS::AM($key)) {
  $full_name = UCS::AM_Name($key);            # long descriptive name
  $description = UCS::AM_Description($key);   # optional multi-line text
  $exp = UCS::AM_Expression($key);            # AM equation as compiled UCS expression
  $score = $exp->eval({f=>$f, f1=>$f1, ...}); # use UCS::Expression methods to evaluate AM
}
$score = UCS::Eval_AM($key, $arghash);        # convenient but slow

UCS::Load_AM_Package("HTest", ...);           # load built-in AM packages

$ok = UCS::Register_AM                        # register new association measure
  "tscore",                                   # AM key (-> variables am.tscore and r.tscore)
  "t-score measure (Church et. al. 1991)",    # long descriptive name
  '(%O11% - %E11%) / sqrt(%O11%)',            # UCS expression (will be compiled into UCS::Expression)
  $multiline_text;                            # optional multi-line description of AM
```

DESCRIPTION

This UCS core library maintains a list of **bulit-in AM**s and Perl subroutines for computing their **score**s from a candidate's signatures. Utility functions perform syntax checks for **field name**s, determine **field type**s from the naming conventions, and **match pattern**s containing UCS wildcards against field names.

CONFIGURATION VARIABLES

$UCS::Version;

The currently installed UCS version.

$UCS::Copyright;

A copyright string for the UCS system. Will be displayed by some UCS/Perl scripts.

$UCS::BaseDir;

The base directory of the UCS System installation. Compiled UCS **programs** and links to Perl scripts are installed in *$UCS::BaseDir*/bin/, while the components of **UCS/R** can be found in *$UCS::BaseDir*/R/.

$UCS::PerlDir;

The base directory of the **UCS/Perl** installation. The UCS Perl modules are installed in *$UCS::PerlDir*/lib/ and its subdirectories, Perl scripts in *$UCS::PerlDir*/bin/.

GENERAL FUNCTIONS

UCS::Die($message, ...);

"Safe" replacement for Perl's built-in **die** function, which will even exit properly from a Perl/Tk loop. One or more lines of error messages are printed on STDERR (or shown in some other suitable manner).

UCS::Warn($message, ...);

By default, prints one or more lines of warning/error messages on STDERR like **UCS::Die**, but does not exit the script. The purpose of this replacement for the built-in **warn** function is to allow warnings to be caught and displayed in a Perl/Tk user interface. Warnings might also be redirected to a log file.

UCS::Status($message);

Displays a status message in a Perl/Tk interface. By default, *$message* is appended to any previous messages. When *$message* ends in a newline character (\n), the next call to **UCS::Status** will replace the current message; when it ends in a carriage return (\r), the next call will overwrite the current message from the start. (This is the usual effect of **print**ing such control characters, and will be simulated in Perl/Tk interfaces).

UCS::Splash();

Displays a UCS splash screen with UCS version information and copyright, e.g. during the start-up phase of a larger UCS/Perl script.

$UCS::Verbose = 0;

The variable *$UCS::Verbose* controls whether status messages and warnings are printed on STDOUT and STDERR, respectively. Verbose output is enabled by default, and can be suppressed by setting *$UCS::Verbose* to 0.

@unique_values = UCS::Unique(@list);

Removes duplicate values from *@list* and returns the remaining elements in the original order. Useful to avoid repretitions of variable names etc.

MANIPULATING VARIABLE NAMES

$std_vars = (@UCS::CoreVars, @UCS::DerivedVars);

Names of **core** and **derived variables**.

$ok = UCS::ValidKey($key);

Returns true iff *$key* is a valid UCS identifier, which may be used as an AM key or in the name of a user-defined variable.

$ok = UCS::ValidName($name);

Returns true iff *$name* is a valid UCS variable name, i.e. either a standard variable (core or derived) , an association score or ranking, or a user-defined variable. See *ucsfile* for details on the UCS naming conventions.

$type = UCS::VarType($name);

Determines the data type of a variable from its name *$name*, according to the UCS naming conventions. Possible data types are `BOOL` (Boolean, 0/1), `INT` (signed integer), `DOUBLE` (double-precision floating-point), and `STRING` (string value).

($spec, $key) = UCS::SplitName($name);

Splits the variable name *$name* of an association score, ranking, or user-defined variable into the specifier *$spec* and the key *$key*. *$spec* will be one of `am`, `r`, `b`, `f`, `n`, or `x`. If *$name* is invalid or the name of a standard variable, (undef, *$name*) is returned.

@matches = UCS::Match($pattern, @names);

Extract strings from *@names* that match the UCS **wildcard pattern** *$pattern*. The pattern may contain literal characters `A-Z a-z 0-9 .` and the wildcards `?`, `*`, and `%`.

```
?  ...  arbitrary character
*  ...  arbitrary substring without "."
%  ...  arbitrary string
```

Thus, the pattern `%` selects all field names, `*` selects the names of core and derived fields, `am.%` all AM scores, etc. See *ucsexp* for more examples.

ASSOCIATION MEASURE REGISTRY

This **registry** maintains a list of association measures, which are automatically available to all UCS/Perl scripts. Association measures are identified by their **key**, which must be a valid UCS identifier. Association scores for a measure with the key `fisher`, for instance, will be stored in the variable `am.fisher`, and the corresponding rankings in the variable `r.fisher`. A wide range of predefined association measures can be imported from the **UCS::AM** module and several add-on packages (see the UCS::AM manpage).

@registered_AMs = UCS::AM_Keys();

The **UCS::AM_Keys** function returns the keys of all currently registered association measures as an unordered list. (Note that no association measures are defined unless **UCS::AM** and/or the add-on packages have been imported.)

$ok = UCS::AM($key);

Returns true if an association measure is registered under *$key*.

$full_name = UCS::AM_Name($key);

Returns a long and descriptive name for the association measure identified by *$key*. This name should be suitable for presentation to the user in a selection dialogue.

$description = UCS::AM_Description($key);

An optional lengthy description of the association measure identified by *$key*. *$description* is a single string but will usually contain linebreaks (`\n`), which may need to be removed for automatic justification (e.g. in a Perl/Tk interface).

$exp = UCS::AM_Expression($key);

Returns the equation of the association measure *$key*, compiled into a **UCS::Expression** object. Call the **eval** or **evalloop** method of *$exp* to compute association scores (see *UCS::Expression*). The sourcecode of this expression can be retrieved with the **string** method (which is especially useful for built-in association measures).

$score = UCS::Eval_AM($key, $arghash);

The **UCS::Eval_AM** function is a convenient and shorter alternative, and is equivalent to:

```
$exp = UCS::AM_Expression($key);
$score = $exp->eval($arghash);
```

It incurs considerable overhead when association scores are calculated for multiple pair types (because of the repeated lookup of *$key* in the AM registry), and should be avoided in tight loops. (See *UCS::Expression* for some comments on efficiency.)

@packages = UCS::Load_AM_Package($name, ...);

Load one or more of the built-in AM packages as specified by the function arguments. *$name* must match the last part of the corresponding module name, e.g. `'HTest'` to load the **UCS::AM::HTest** package. *$name* is case-insensitive and may be abbreviated to a unique prefix. The special name `'ALL'` (or `'all'`) loads all available add-on packages, while the empty string `''` loads the basic measures from **UCS::AM**. **UCS::Load_AM_Package** returns a list containing the full names of all loaded packages (with duplicates removed). If there is no match for *$name*, an empty list is returned.

$ok = UCS::Register_AM($key, $name, $equation [, $description]);

The **UCS::Register_AM** function is used to register a new association measure, or overwrite an existing one with a new definition. *$key* is the identification key of the new measure, *$name* a descriptive name, *$equation* the measure's equation in the form of an (uncompiled) UCS expression, and *$description* an optional multi-line description. *$equation* may also be an object of class **UCS::Expression** (which is cloned rather than re-compiled), enabling the use of advanced features such as parametric expressions.

The function call returns true if the new measure has been successfully registered. A false return value indicates that compilation of *$equation* into an **UCS::Expression** object failed. The **UCS::Register_AM** function will **die** if *$key* is not a valid UCS identifier.

The example below shows the code used to register the **t-score** measure (Church *et. al.* 1991) which has been widely used in English lexicography.

```
$ok = UCS::Register_AM  "tscore",
  "t-score measure (Church et. al. 1991)",
  '(%O11% - %E11%) / sqrt(%O11%)',
  "The t-score measure applies Student's t-test to ...";
die "Syntax error in UCS expression for t-score measure"
  unless $ok;
```

SEE ALSO

Type `ucsdoc ucsintro` for an introduction to UCS/Perl and an overview of its components (in the MODULES and PROGRAMS sections).

COPYRIGHT

3.2 UCS::File

File access utilities

SYNOPSIS

```
use UCS::File;

## open filehandle for reading or writing
# automagically compresses/decompresses files and dies on error
$fh = UCS::File::Open("> my_file.gz");
# the same without error checks (may return undefined value)
$fh = UCS::File::TryOpen("> my_file.bz2");

## temporary file objects (disk files are automatically removed)
$t1 = new UCS::File::Temp;             # picks a unique filename
$t2 = new UCS::File::Temp "mytemp";    # extends prefix to unique name
$t3 = new UCS::File::Temp "mytemp.gz"; # compressed temporary file
$filename = $t1->name;        # full pathname of temporary file
$t1->write(...);              # works like $fh->print() ;
$t1->finish;                  # stop writing file
print $t1->status, "\n";      # WRITING/FINISHED/READING/DELETED
# main program can read or overwrite file <$filename> now
$line = $t1->read;            # read one line (like $fh->getline())
$t1->rewind;                  # re-read from beginning of file
$line = $t1->read;            # (reads first line again)
$t1->close;                   # stop reading and remove temporary file
# other files will be removed when objects $t2 and $t3 are destroyed

## execute shell command with error detection
$cmd = "ls -l";
$errlevel = UCS::File::ShellCmd($cmd); # dies with error message if not ok
$UCS::File::Paranoid = 1;     # more paranoid checks (-1 for less paranoid)
# $errlevel == 0 (ok), 1 (minor problems), ..., 6 (fatal error)

UCS::File::ShellCmd($cmd, \@lines);    # capture standard output in array
UCS::File::ShellCmd($cmd, "file.txt"); # ... or in file (for large amounts of data)
UCS::File::ShellCmd(["ls", "-l", @files], \@lines);  # bypass shell expansion
```

DESCRIPTION

This module provides some useful routines for handling files and external programs. This includes **opening files** with error checks and automagical compression/decompression, **temporary file objects** that are automatically created and deleted, and the execution of **shell commands** with extensive error checks.

OPENING FILES

$fh = UCS::File::Open($name);

Open file *$name* for reading, writing, or appending. Returns **FileHandle** object if successful, otherwise it **die**s with an error message. It is thus never necessary to check whether *$fh* is defined.

If *$name* starts with >, the file is opened for writing (an existing file will be overwritten). If *$name* starts with >>, the file is opened for appending.

Files with the extensions `.Z`, `.gz`, and `.bz2` are automagically compressed and decompressed, provided that the necessary tools are installed. It is also possible to append to `.gz` and `.bz2` files.

Note that *$name* may also be a read or write pipe ("... |" or "| ...", respectively), which is passed directly to the built-in **open** command. It is thus subject to shell expansion and does not support automagic compression and decompression.

$fh = UCS::File::TryOpen($name);

Same as **UCS::File::Open**, but without the error checks. Returns **undef** if the `open()` call fails.

TEMPORARY FILES

Temporary files (implemented by **UCS::File::Temp** objects) are assigned a unique name and are automatically deleted when the script exits. The life cycle of a temporary file consists of four stages: **create**, **write**, **read** (possibly **re-read**), **delete**. This cycle corresponds to the following method calls:

```
 $tf = new UCS::File::Temp; # create new temporary file in /tmp dir
 $tf->write(...);     # write cycle (buffered output, like print function)
 $tf->finish;         # complete write cycle (flushes buffer)
 $line = $tf->read;   # read cycle (like getline method for FileHandle)
[$tf->rewind;         # optional: start re-reading temporary file ]
[$line = $tf->read;                                              ]
 $tf->close;          # delete temporary file
```

Once the temporary file has been read from, it cannot be re-written; a new **UCS::File::Temp** object has to be created for the next cycle. When the write stage is completed (but before reading has started, i.e. after calling the **finish** method), the temporary file can be accessed and/or overwritten by external programs. Use the **name** method to obtain its full pathname. If no direct access to the temporary file is required, the **finish** method is optional. The write cycle will automatically be completed before the first **read** method call.

$tf = new UCS::File::Temp [$prefix ;]

Creates temporary file in */tmp* directory. If the optional argument *$prefix* is specified, the filename will begin with *$prefix* and be extended to a unique name. If *$prefix* contains a `/` character, it is interpreted as an absolute or relative path, and the temporary file will not be created in the */tmp* directory. To create a temporary file in the current working directory, use `./MyPrefix`.

You can add the extension `.Z`, `.gz`, or `.bz2` to *$prefix* in order to create a compressed temporary file. The actual filename (as returned by the **name** method) will have the same extension in this case.

The temporary file is immediately created and opened for writing.

$filename = $tf->name;

Returns the real filename of the temporary file. **NB:** direct access to this file (e.g. by external programs) is only allowed after calling **finish**, and before the first **read**.

$tf->write(...);

Write data to the temporary file. All arguments are passed to Perl's built-in **print** function. Like **print**, this method does not automatically add newlines to its arguments.

$tf->finish;

Stop writing to the temporary file, flush the output buffer, and close the associated file handle. Afer **finish** has been called, the temporary file can be accessed directly by the script or external programs, and may also be overwritten. In order to delete a file created by an external program automatically, **finish** the temporary file immediately after its creation and then allow the external tool to overwrite it:

```
$tf = new UCS::File::Temp;
$tf->finish;  # temporary file has size of 0 bytes now
$filename = $tf->name;
system "$my_shell_command > $filename";
```

$line = $tf->read;

Read one line from temporary file (same as calling **getline** on a **FileHandle** object). Automatically invokes **finish** if called during write cycle.

$tf->rewind;

Allows re-reading of the temporary file. The next **read** call will return the first line of the temporary file. Internally this is achieved by closing and re-opening the associated file handle.

$tf->close;

Closes any open file handles and deletes the temporary file. This will be done automatically when the **UCS::File::Temp** object is destroyed. Use **close** to free disk space immediately.

SHELL COMMANDS

The **UCS::File::ShellCmd** function provides a convenient replacement for the built-in **system** command. Standard output and error messages produced by the invoked shell command are captured to avoid screen clutter. The collected standard ouput of the command can optionally be returned to the caller (similar to the backtick operator `` `$shell_cmd` ``). **UCS::File::ShellCmd** also checks for a variety of error conditions and returns an error level ranging from 0 (successful) to 6 (fatal error):

```
Error Level  Description
  6          command execution failed (system error)
  5          non-zero exit value or error message on STDERR
  4          -- reserved for future use --
  3          warning message on STDERR
  2          any output on STDERR
  1          error message on STDOUT
```

Depending on the value of *$UCS::File::Paranoid* and the error level, a warning message may be issued or the function may **die** with an error message.

$UCS::File::Paranoid = 0;

With the default setting of 0, **UCS::File::ShellCmd** will **die** if the error level is 5 or greater. In the **extra paranoid** setting (+1), it will almost always **die** (error level 2 or greater). In the **less paranoid** setting (-1) only an error level of 6 (i.e. failure to execute the shell command) will cause the script to abort.

$errlvl = UCS::File::ShellCmd($cmd);

$errlvl = UCS::File::ShellCmd($cmd, $filename);

$errlvl = UCS::File::ShellCmd($cmd, \@lines);

The first form executes *$cmd* as a shell command (through the built-in **system** function) and returns an error level. With the default setting of *$UCS::File::Paranoid*, serious errors are usually detected and cause the script to **die**, so it is not necessary to check the value of *$errlvl*.

The second form stores the standard output of the shell command in a file named *$filename*, where it can then be processed with external programs or read in by the Perl script. **NB:** Compressed files are not supported! It is recommended to use an uncompressed temporary file (**UCS::File::Temp** object).

The third form takes an array reference as its second argument, splits the standard output of *$cmd* into **chomp**ed lines and stores them in the array *@lines*. If there is a large amount of standard ouput, it is more efficient to use the second form.

Note that *$cmd* is passed to the shell for metacharacter expansion. In order to avoid this (e.g. when filename arguments may contain blanks), specify an array reference of the form `[$program, @args]` instead:

```
$errlvl = UCS::File::ShellCmd(["ls", "-l", @files], \@lines);
```

COPYRIGHT

3.3 UCS::R

UCS/Perl interface to R

SYNOPSIS

```
use UCS::R;

UCS::R::Start();               # start R backend explicitly
UCS::R::Stop();                # terminate R backend (if possible)

@x = UCS::R::Exec($cmd);       # execute R cmd (must return numeric vector)

UCS::R::LoadVector("my.x", \@data);  # load numeric vector efficiently into R
$data = UCS::R::DumpVector("my.x");  # returns arrayref

# access to special functions and statistical distributions
# through the UCS::SFunc module
```

DESCRIPTION

The **UCS::R** module provides an interface to the **R** statistical environment and the **UCS/R** libraries on an R interpreter running in the background. When available (as determined by the installation script), the **RSPerl** interface is used for efficient communication with the R interpreter. Otherwise, the system falls back on a slower but more portable solution that simulates an interactive R session through use of the **Expect** module. See the UCS::R::RSPerl and UCS::R::Expect manpages for some details about the strengths and limitations of the two backends.

The **UCS::R** interface is mainly used by the **UCS::SFunc** module to make the R implementations of **special functions** (binomial coefficients, Gamma function, Beta function) and **statistical distributions** (binomial, Poisson, normal, chi-squared, hypergeometric) available to **UCS/Perl**, without relying on an external maths library and/or compiled C code.

FUNCTIONS

UCS::R::Start();

Starts the **R** interpreter. Normally, this function does not have to be called explicitly, as the backend is automatically launched when an R command is executed for the first time. Since this will block program execution for a few seconds, some scripts may prefer to call **UCS::R::Start** at start-up time before the R process is actually needed.

UCS::R::Stop();

Terminate the **R** interpreter. Normally, this function does not have to be called explicitly, but it may be used to shut down an R process that is no longer needed and free memory resources. Note that this function is not supported by the **UCS::R::RSPerl** backend and will be silently ignored.

@x = UCS::R::Exec($cmd);

Executes the **R** command *$cmd* in the server process. The command must return a vector, which is passed back to the calling script in the form of a list *@x*. When command execution fails or its return value cannot be parsed, the **UCS::R::Exec** function will **die** with an error message.

At the moment, only numeric vectors are guaranteed to work (although the **UCS::R::RSPerl** backend should support all types of vectors). It is safe to execute any command when **UCS::R::Exec** is called in void context. When using the **UCS::R::Expect** backend, complex return values should be made **invisible** for reasons of speed and robustness.

NB: This interface is not efficient for exchanging large amounts of data with R and may hang if the input/output buffers overflow. Use the **LoadVector** and **DumpVector** functions for this purpose (see below). Moreover, *$cmd* must be a single-line command (separate

multiple commands with `;`), so that it leaves a single command prompt at the beginning of a line after execution. Avoid `cat()` and any functions that prompt for user input, otherwise **UCS::R::Exec** will become confused and may hang.

UCS::R::LoadVector($varname, \@data);

Efficiently loads a numeric vector into **R** (making use of a temporary file and the `scan` function in R). The data *@data* are passed in as an array reference and will be stored in the **R** variable *$varname*.

$data = UCS::R::DumpVector($varname);

Efficiently reads a numeric vector from **R** (making use of a temporary file and the `write()` function). The data stored in the **R** variable *$varname* (which must be a numeric vector) are returned as an anonymous array reference *$data*.

SPECIAL FUNCTIONS AND STATISTICAL DISTRIBUTIONS

The special functions and statistical distributions provided through the **R** interface are not exported by this module. Use **UCS::SFunc** instead. All available functions are documented in the UCS::SFunc manpage. They are available under the same names in the **UCS::R** package. For instance, the R implementation of the **lgamma** function can be accessed explicitly as **UCS::R::lgamma**.

COPYRIGHT

3.4 UCS::R::Expect

Expect-based implementation of R backend

SYNOPSIS

```
use UCS::R::Expect;
## exports Start(), Stop() and Exec() functions into current namespace
## as well as LoadVector() and DumpVector()
```

DESCRIPTION

This module should *only* be used implicitly through **UCS::R**, which loads the more efficient **UCS::R::RSPerl** implementation if available, and falls back on **UCS::R::Expect** otherwise.

LIMITATIONS

This module starts an R process in the background and communicates with it interactively through the **Expect** module. This approach has several disadvantages:

- Invoking R commands, waiting for output from the R backend, and parsing that output causes substantial overhead for R function invocations, allowing less than 1000 invocations per second even on a fast machine.
- The return value of a function call has to be printed by R, then the resulting output has to be parsed by Perl. This interfacing method is rather frail and currently supports only numeric vectors as return values.
- The interface is extremely inefficient for exchanging large amounts of data between Perl and R. It may hang if the input/output buffers used by **Expect** overflow. Use the **LoadVector** and **DumpVector** functions to pass large numeric vectors to R and back.

Because of these limitations, it is highly recommended that you install and use the **RSPerl** interface (available from http://www.omegahat.org/) on Unix platforms. When RSPerl has been installed with support for calling R from Perl, it will automatically be detected and configured for use by the UCS installation script. See *doc/install.txt* for more information and installation tips.

COPYRIGHT

3.5 UCS::R::RSPerl

RSPerl-based implementation of R backend

SYNOPSIS

```
use UCS::R::RSPerl;
## exports Start(), Stop() and Exec() functions into current namespace
## as well as LoadVector() and DumpVector()
```

DESCRIPTION

This module should *only* be used implicitly through **UCS::R**, which loads the **UCS::R::RSPerl** implementation if available, and falls back on the inefficient **UCS::R::Expect** implementation otherwise.

Note that `use UCS::R::RSPerl` will fail if RSPerl support is not available, causing the compilation of the Perl script to abort.

COPYRIGHT

3.6 UCS::SFunc

Special functions and statistical distributions

SYNOPSIS

```
use UCS::SFunc;

# special functions (all logarithms are base 10)
$c     = choose($n, $k);       # binomial coefficient
$log_c = lchoose($n, $k);

$y     = gamma($a);            # Gamma function
$log_y = lgamma($a);
$y     = igamma($a, $x [, $upper]);  # incomplete Gamma functions
$log_y = ligamma($a, $x [, $upper]);
$y     = rgamma($a, $x [, $upper]);  # regularised Gamma functions
$log_y = lrgamma($a, $x [, $upper]);
$x     = igamma_inv($a, $y [, $upper]); # inverse Gamma functions
$x     = ligamma_inv($a, $log_y [, $upper]);
$x     = rgamma_inv($a, $y [, $upper]);
$x     = lrgamma_inv($a, $log_y [, $upper]);

$y     = beta($a, $b);         # Beta function
$log_y = lbeta($a, $b);
$y     = ibeta($x, $a, $b);    # incomplete Beta function
$log_y = libeta($x, $a, $b);
$y     = rbeta($x, $a, $b);    # regularised Beta function
$log_y = lrbeta($x, $a, $b);
$x     = ibeta_inv($y, $a, $b);        # inverse Beta functions
$x     = libeta_inv($log_y, $a, $b);
$x     = rbeta_inv($y, $a, $b);
$x     = lrbeta_inv($log_y, $a, $b);

# binomial distribution (density, tail probabilities, quantiles)
$d  = dbinom($k, $size, $prob);
$ld = ldbinom($k, $size, $prob);
$p  = pbinom($k, $size, $prob [, $upper]);
$lp = lpbinom($k, $size, $prob [, $upper]);
$k  = qbinom($p, $size, $prob [, $upper]);
$k  = lqbinom($lp, $size, $prob [, $upper]);

# Poisson distribution (density, tail probabilities, quantiles)
$d  = dpois($k, $lambda);
$ld = ldpois($k, $lambda);
$p  = ppois($k, $lambda [, $upper]);
$lp = lppois($k, $lambda [, $upper]);
$k  = qpois($p, $lambda [, $upper]);
$k  = lqpois($lp, $lambda [, $upper]);

# normal distribution (density, tail probabilities, quantiles)
$d  = dnorm($x, $mu, $sigma);
$ld = ldnorm($x, $mu, $sigma);
$p  = pnorm($x, $mu, $sigma [, $upper]);
$lp = lpnorm($x, $mu, $sigma [, $upper]);
$x  = qnorm($p, $mu, $sigma [, $upper]);
$x  = lqnorm($lp, $mu, $sigma [, $upper]);
```

```
# chi-squared distribution (density, tail probabilities, quantiles)
$d  = dchisq($x, $df);
$ld = ldchisq($x, $df);
$p  = pchisq($x, $df [, $upper]);
$lp = lpchisq($x, $df [, $upper]);
$x  = qchisq($p, $df [, $upper]);
$x  = lqchisq($lp, $df [, $upper]);

# hypergeometric distribution (density and tail probabilities)
$d = dhyper($k, $R1, $R2, $C1, $C2);
$ld = ldhyper($k, $R1, $R2, $C1, $C2);
$p = phyper($k, $R1, $R2, $C1, $C2 [, $upper]);
$lp = lphyper($k, $R1, $R2, $C1, $C2 [, $upper]);
```

DESCRIPTION

This module provides **special functions** and common **statistical distributions**. Currently, all functions are imported from the **UCS/R** system (using the **UCS::R** interface).

SPECIAL FUNCTIONS

UCS::SFunc currently provides the following special mathematical functions: **binomial coefficients**, the **Gamma function**, the **incomplete Gamma functions** and their inverses, the **regularised Gamma functions** and their inverses, the **Beta function**, the **incomplete Beta function** and its inverse, and the **regularised Beta function** and its inverse. Note that all logarithmic versions return **base 10** logarithms!

$coef = choose($n, $k);

$log_coef = lchoose($n, $k);

The **binomial coefficient** "*$n* over *$k*", and its logarithm.

$y = gamma($a);

$log_y = lgamma($a);

The (complete) **Gamma function** with argument *$a*, and its logarithm. Note that the factorial *n!* is equal to `gamma(`*n*`+1)`.

$y = igamma($a, $x [, $upper);]

$log_y = ligamma($a, $x [, $upper);]

The **incomplete Gamma function** with arguments *$a* and *$x*, and its logarithm. If *$upper* is specified and true, the upper incomplete Gamma function is computed, otherwise the lower incomplete Gamma function. It is recommended to set *$upper* to the string constant `'upper'` as a reminder of its function.

$x = igamma_inv($a, $y [, $upper);]

$x = ligamma_inv($a, $log_y [, $upper);]

The **inverse** of the **incomplete Gamma function**, as well as the inverse of its logarithm.

$y = rgamma($a, $x [, $upper);]

$log_y = lrgamma($a, $x [, $upper);]

The **regularised Gamma function** with arguments *$a* and *$x*, and its logarithm. If *$upper* is specified and true, the upper regularised Gamma function is computed, otherwise the lower regularised Gamma function. It is recommended to set *$upper* to the string constant `'upper'` as a reminder of its function.

$x = rgamma_inv($a, $y [, $upper);]

$x = lrgamma_inv($a, $log_y [, $upper);]

The **inverse** of the **regularised Gamma function**, as well as the inverse of its logarithm.

$beta = beta($a, $b);

$log_beta = lbeta($a, $b);

The (complete) **Beta function** with arguments *$a* and *$b*, and its logarithm.

$y = ibeta($x, $a, $b);

$log_y = libeta($x, $a, $b);

The **incomplete Beta function** with arguments *$x*, *$a*, and *$b*, and its logarithm.

$x = ibeta_inv($y, $a, $b);

$x = libeta_inv($log_y, $a, $b);

The **inverse** of the **incomplete Beta function**, as well as the inverse of its logarithm.

$y = rbeta($x, $a, $b);

$log_y = lrbeta($x, $a, $b);

The **regularised Beta function** with arguments *$x*, *$a*, and *$b*, and its logarithm.

$x = rbeta_inv($y, $a, $b);

$x = lrbeta_inv($log_y, $a, $b);

The **inverse** of the **regularised Beta function**, as well as the inverse of its logarithm.

STATISTICAL DISTRIBUTIONS

UCS::SFunc computes **densities**, **tail probabilities** (= distribution function), and **quantiles** for the following statistical distributions: **binomial** distribution, **Poisson** distribution, **normal** distribution, **chi-squared** distribution, **hypergeometric** distribution. The function names are the common abbreviations as used e.g. in the **R** language, with additional logarithmic versions (that start with the letter `l`) (these correspond to the `log=TRUE` and `log.p=TRUE` parameters in R).

Note that logarithmic probabilities are always given as **negative base 10** logarithms. The logarithmic density and tail probability functions return such logarithmic p-values, and the quantile functions expect them in their first argument.

The Binomial Distribution Binomial distribution with parameters *$size* (= number of trials) and *$prob* (= success probability in single trial). *E[X] = $size * $prob*, *V[X] = $size * $prob * (1 - $prob)*.

$d = dbinom($k, $size, $prob);

$ld = ldbinom($k, $size, $prob);

Density *P(X = $k)* and its negative base 10 logarithm.

$p = pbinom($k, $size, $prob [, $upper);]

$lp = lpbinom($k, $size, $prob [, $upper);]

Tail probabilities *P(X <= $k)* and *P(X > $k)* (if *$upper* is specified and true), and their negative base 10 logarithms. It is recommended to set *$upper* to the string `'upper'` as a reminder of its meaning.

The R implementation of binomial tail probabilities underflows for very small probabilities (even in the logarithmic version), as of R version 2.1. Therefore, these functions use a mixture of R and Perl code to compute upper tail probabilities for large samples (which are most likely to lead to undeflow problems for cooccurrence data).

$k = qbinom($p, $size, $prob [, $upper);]

$k = lqbinom($lp, $size, $prob [, $upper);]

Lower and upper quantiles. The lower quantile is the smallest value *$k* with *P(X <= $k) >= $p*. The upper quantile (which is computed when *$upper* is specified and true) is the largest value *$k* with *P(X > $k) >= $p*. In the logarithmic version, *$lp* must be the negative base 10 logarithm of the desired p-value.

Note that these functions use the R implementation directly without a workaround for underflow problems. The quantiles returned for very small p-values (especially when using **lqbinom**) are therefore unreliable and should be used with caution.

The Poisson Distribution Poisson distribution with parameter *$lambda* (= expectation); *E[X] = V[X] = $lambda*.

$d = dpois($k, $lambda);

$ld = ldpois($k, $lambda);

Density *P(X = $k)* and its negative base 10 logarithm.

$p = ppois($k, $lambda [, $upper);]

$lp = lppois($k, $lambda [, $upper);]

Tail probabilities *P(X <= $k)* and *P(X > $k)* (if *$upper* is specified and true), and their negative base 10 logarithms. It is recommended to set *$upper* to the string `'upper'` as a reminder of its meaning.

$k = qpois($p, $lambda [, $upper);]

$k = lqpois($lp, $lambda [, $upper);]

Lower and upper quantiles. The lower quantile is the smallest value *$k* with *P(X <= $k) >= $p*. The upper quantile (which is computed when *$upper* is specified and true) is the largest value *$k* with *P(X > $k) >= $p*. In the logarithmic version, *$lp* must be the negative base 10 logarithm of the desired p-value.

The Normal Distribution Normal distribution with parameters *$mu* (= expectation) and *$sigma* (= standard deviation). Unspecified parameters default to *$mu = 0* and *$sigma = 1*. *E[X] = $mu*, *V[X] = $sigma ** 2*.

$d = dnorm($x, $mu, $sigma);

$ld = ldnorm($x, $mu, $sigma);

Density *P(X = $x)* and its negative base 10 logarithm.

$p = pnorm($x, $mu, $sigma [, $upper);]

$lp = lpnorm($x, $mu, $sigma [, $upper);]

Tail probabilities *P(X <= $x)* and *P(X >= $x)* (if *$upper* is specified and true), and their negative base 10 logarithms. It is recommended to set *$upper* to the string `'upper'` as a reminder of its meaning.

$x = qnorm($p, $mu, $sigma [, $upper);]

$x = lqnorm($lp, $mu, $sigma [, $upper);]

Lower and upper quantiles. The lower quantile is the smallest value *$x* with *P(X <= $x) >= $p*. The upper quantile (which is computed when *$upper* is specified and true) is the largest value *$x* with *P(X >= $x) >= $p*. In the logarithmic version, *$lp* must be the negative base 10 logarithm of the desired p-value.

The Chi-Squared Distribution Chi-squared distribution with parameter *$df* (= degrees of freedom); *E[X] = $df, V[X] = 2 * $df.*

$d = dchisq($x, $df);

$ld = ldchisq($x, $df);

Density function *f(x)* and its negative base 10 logarithm.

$p = pchisq($x, $df [, $upper);]

$lp = lpchisq($x, $df [, $upper);]

Tail probabilities *P(X <= $x)* and *P(X >= $x)* (if *$upper* is specified and true), and their negative base 10 logarithms. It is recommended to set *$upper* to the string `'upper'` as a reminder of its meaning.

$x = qchisq($p, $df [, $upper);]

$x = lqchisq($lp, $df [, $upper);]

Lower and upper quantiles. The lower quantile is the smallest value *$x* with *P(X <= $x) >= $p*. The upper quantile (which is computed when *$upper* is specified and true) is the largest value *$x* with *P(X >= $x) >= $p*. In the logarithmic version, *$lp* must be the negative base 10 logarithm of the desired p-value.

The Hypergeometric Distribution Hypergeometric distribution of the upper left-hand corner X in a 2x2 contingency table with fixed marginals *$R1*, *$R2*, *$C1*, and *$C2*, where both *$R1 + $R2* and *$C1 + $C2* must sum to the sample size *N*. *$k* represents the observed value of X and must be in the admissible range *max(0, $R1 - $C2) <= $k <= min($R1, $C1)*, otherwise the density will be given as 0 and tail probabilities as 1 or 0, respectively. *E[X] = $R1 * $C1 / $N*, *V[X] = $R1 * $R2 * $C1 * $C2 / (N^2 * (N-1))*.

For R versions before 2.0, the upper tail probabilities are computed with a mixture of R and Perl code to circumvent a cancellation problem in the R implementation and achieve better precision. For this reason, the functions for quantiles are currently not supported (but may be when R version 2.0 is required for the UCS toolkit).

$d = dhyper($k, $R1, $R2, $C1, $C2);

$ld = ldhyper($k, $R1, $R2, $C1, $C2);

Density *P(X = $k)* and its negative base 10 logarithm.

$p = phyper($k, $R1, $R2, $C1, $C2 [, $upper]);

$lp = lphyper($k, $R1, $R2, $C1, $C2 [, $upper]);

Tail probabilities *P(X <= $k)* and *P(X > $k)* (if *$upper* is specified and true), and their negative base 10 logarithms. It is recommended to set *$upper* to the string `'upper'` as a reminder of its meaning.

COPYRIGHT

3.7 UCS::Expression

Compile and execute UCS expressions

SYNOPSIS

```
use UCS::Expression;

$exp = new UCS::Expression $code;  # compile UCS expression
@vars = $exp->needed;              # variables needed to evaluate expression
$code = $exp->string;              # retrieve sourcecode of UCS expression
$result = $exp->eval(@args);       # evaluate UCS expression (argument list)
$result = $exp->eval($arghash);    # named arguments (UCS variable names)

$exp = new UCS::Expression $code, "MU" => 10, ...; # expression with parameters
@params = $exp->params;            # sorted list of parameter names
$value = $exp->param("MU");        # current value of parameter
$exp->set_param("MU", 1);          # set parameter value
$exp2 = $exp->copy;                # clone expression (e.g. when changing parameters)

$sub = $exp->code;                 # reference to compiled Perl expression
$result = $sub->(@args);           # argument list is same as for eval()

$listref = $exp->evalloop($size, $arghash);  # evaluate expression on full data set
$exp->evalloop(\@result, $size, $arghash);   # directly writes to array @result
```

DESCRIPTION

UCS expressions provide a convenient way to evaluate functions and conditions on the pair types in a data set. They consist of arbitrary Perl code with a syntax extension for direct access to data set variables: the character sequence *%varname%* (where *varname* is a legal UCS variable name) is replaced by the value of this variable (for the current pair type). See *ucsexp* for a more detailed description of UCS expressions and some cautionary remarks.

A **UCS::Expression** object represents a compiled UCS expression. The **needed** method returns a list of UCS variables that are required for evaluation of the expression. When **derived variables** are used in a UCS expression, they are automatically computed from the frequency signature.

The **eval** method is normally invoked with a (reference to a) hash of arguments, using UCS variable names as keys. It selects the variables needed to evaluate the UCS expression automatically from the hash, and ensures that all of them are present. Better performance is achieved by passing the required variables as an argument list in the correct order (as returned by **needed**).

The **evalloop** method greatly reduces overhead when a UCS expression is applied to a list of pair types (i.e. a full data set). It expects array references instead of simple variable values, and returns a reference to an array of the specified length. Optionally, **evalloop** can write directly to an existing array.

METHODS

$exp = new UCS::Expression $code;

Compiles the UCS expression *$code* into a **UCS::Expression** object. If compilation fails for some reason, an **undef**ined value is returned. Compiling a UCS expression involves the follwing steps:

- All UCS variable references in *$code* are identified and validated.
- A list of required variables is constructed. Derived variables are implicitly computed from the frequency signature, and the necessary core variables are automatically added to the list of required variables.
- The UCS variable references are substituted with lexical Perl variables, which are initialised from the parameter list `@_`.

- The resulting Perl code is compiled into an anonymous subroutine, which is stored in the **UCS::Expression** object and can be executed through the **eval** method.

Since **UCS::Expressions** are comparatively small structures, it is usually not necessary to destroy them explicitly.

$exp = new UCS::Expression $code, $param => $value, ...;

This form of the constructor defines a UCS expression with parameters, given as pairs of parameter name *$param* and default value *$value*. Parameters can be used like variables in the UCS expression. Their names are simple UCS identifiers, but **must not** be valid UCS variable names. The recommended convention is to write parameter names all in upper case.

@names = $exp->params;

Returns the names of all parameters in alphabetical order.

$value = $exp->param($name);

Returns the current value of parameter *$name*;

$exp->set_param($name, $value);

Set the parameter *$name* to the value *$value*. The new value will be used by all subsequent calls to the **eval** and **evalloop** methods.

$new_exp = $exp->copy;

Makes a clone of the **UCS::Expression** object *$exp*. Cloning is a fast operation and should always be used when changing the parameters of an expression shared between different modules (e.g. a registered association measure).

@vars = $exp->needed;

The **needed** methods returns a list of UCS variable names, corresponding to the data set variables needed to evaluate *$exp*.

$code = $exp->string;

Returns the original UCS expression represented by *$exp* as a string, and can be used to modify and recompile UCS expressions (especially those of built-in association measures). Note that *$code* is **chomp**ed, but may contain internal linebreaks (`\n`).

$result = $exp->eval($arghash);

The **eval** method evaluates a compiled UCS expression on the data passed in *$arghash*, which must be a reference to a hash of variable names and the corresponding variable values. The necessary variables are extracted from *$arghash* by name, and the method **die**s with an error message unless all required variables are present. Unused variables are silently ignored.

$result = $exp->eval(@args);

The second form of the **eval** method avoids the overhead of variable name lookup and error checking. Here, the argument list *@arg* consists of the values of all required variables in the order defined by the **needed** method. The list *@args* is passed directly to the compiled Perl code, so that errors will usually go undetected.

$sub = $exp->code;

The **code** method returns a code reference to the anonymous subroutine that resulted from compilation of the UCS expression. For an expression without parameters, the subroutine call

```
$result = $sub->(@args);
```

is equivalent to

```
$exp->eval(@args);
```

and further reduces overhead (by a small amount). It may be useful when the UCS expression is repeatedly applied, looping over a list of pair types. In most such cases, the **evalloop** method provides a better solution, though.

$listref = $exp->evalloop($size, $arghash);

$exp->evalloop(\@result, $size, $arghash);

The **evalloop** method is used to apply *$exp* to an entire list of pair types (i.e. a data set) with a single call. Its invocation is similar to the firs form of the **eval** method. The additional parameter *$size* specifies the number of pair types to be processed. Each value in *$arghash* must be a reference to an array of length *$size*. The return value is a reference to an array of the same length.

The three-parameter form allows **evalloop** to write the results directly into an existing array, which may save a considerable amount of overhead when *$size* is large.

SEE ALSO

See the ucsexp manpage for an introduction to UCS expressions, as well as the UCS::SFunc and UCS::Expression::Func manpages for pre-defined functions that may be used in UCS expressions.

COPYRIGHT

3.8 UCS::Expression::Func

Utility functions for UCS expressions

SYNOPSIS

```
use UCS::Expression::Func;

$min_x = min($x1, $x2, ...);     # minimum of two or more values
$max_y = max(@y);                # maximum of two or more values

$log_prob = -log10($prob);       # base 10 logarithm

$log_prob = inf()                # replace log(Infinity) = -log(0)
  if $prob == 0;                 #   by a very large value
```

DESCRIPTION

This module provides a collection of simple but useful functions, which are automatically imported into the **UCS::Expression** namespace so that they can be used in **UCS expressions** without full qualification.

FUNCTIONS

$min_x = min($x1, $x2, ...);

Minimum of two or more numbers. The argument could also be an array `@x`.

$max_x = max($x1, $x2, ...);

Maximum of two or more numbers. The argument could also be an array `@x`.

$log_prob = -log10($prob);

Base 10 logarithm, which is used for all logarithmic scales in UCS (especially logarithmic p-values). Returns `-inf()` if *$prob* is zero or negative.

$log_infinity = inf();

The **inf** function returns a large positive floating-point value that represents the logarithm of Infinity in UCS/Perl. Note that the logarithm of 0 should consequently be represented by `-inf()`, as does the **log10** function. In order to find out the exact value on your system, you can use the command line

```
ucs-config -e 'print inf(),"\n"'
```

COPYRIGHT

3.9 UCS::AM

Built-in association measures

SYNOPSIS

```
use UCS;
use UCS::AM;

@builtin_AMs = UCS::AM_Keys();

# random
# frequency
# z.score
# z.score.corr
# t.score
# chi.squared
# chi.squared.corr
# log.likelihood
# Poisson.Stirling
# Poisson.pv
# Fisher.pv
# MI
# MI2
# MI3
# relative.risk
# odds.ratio
# odds.ratio.disc
# Dice
# gmean
# MS
# Jaccard
# average.MI
# local.MI
```

DESCRIPTION

This module contains definitions for a wide range of **association measures**. When the **UCS::AM** module is imported, the built-in measures are registered with the **UCS** core library (see *UCS* for details on how to access registered association measures).

The following section gives a full listing of the built-in association measures from the **UCS::AM** module with short explanations. Please refer to *http://www.collocations.de/AM/* for the full equations and references. Further association measures can be imported from **add-on packages** (see the section on §3.9 below).

Note that some association measures produce infinite values (*+inf* or *-inf*). The logarithm of infinity is represented by the return value of the built-in **inf** function (see the UCS::Expression::Func manpage). The association scores of measures with the suffix **.pv** can be interpreted as probabilities (the likelihood of the observed data or the p-value of a statistical hypothesis test). Such probabilities are given as **negative base 10 logarithms**, ranging from 0 to *+inf*. Measures with the suffix **.tt** (for *two-tailed*) are derived from two-sided statistical hypothesis tests. One-sided versions of these tests are provided under the same name, but without the suffix.

BUILT-IN ASSOCIATION MEASURES

random

Random numbers between 0 and 1 as association scores simulate random selection of pair types and are used to break ties when sorting a data set.

frequency

Cooccurrence frequency of the pair type. This association measure is used to sort data sets by frequency, but requires some systematic method for breaking ties.

z.score

A z-score for the observed cooccurrence frequency O11 compared to the expected frequency E11. The value represents a standardised normal approximation of the binomial sampling distribution of O11 under the point null hypothesis of independence.

z.score.corr

A z-score for O11 compared to E11 with Yates' continuity correction applied.

t.score

Church et al (1991) use Student's t-test to compare the observed cooccurrence frequency O11 to the null expectation E11 estimated from the sample (which is a random variate as well), applying several approximations to simplify the **t.score** equation. The computed value is a t-score with degrees of freedom roughly equal to the sample size *N*. This application of the t-test is highly questionable, though, and produces extremely conservative results.

chi.squared

One-sided version of Pearson's chi-squared test for the independence of rows and columns in a 2x2 contingency table. Positive scores indicate positive association ($O11 > E11$), and negative scores indicate negative association ($O11 < E11$). The distinction between positive and negative association is unreliable for small absolute values of the test statistic. Under the null hypothesis, the one-sided **chi.squared** statistic approximates a normal distribution (as the signed root of a chi-squared distribution with one degree of freedom).

chi.squared.corr

One-sided version of Pearson's chi-squared test for the independence of rows and columns in a 2x2 contingency table, with Yates' continuity correction applied.

log.likelihood

One-sided version of the log-likelihood statistic suggested by Dunning (1993), a likelihood ratio test for independence of rows and columns in a 2x2 contingency table (Dunning introduced the measure as a test for homogeneity of the table columns, i.e. equal success probabilites of two independent binomial distributions). Positive scores indicate positive association ($O11 > E11$), and negative scores indicate negative association ($O11 < E11$). The distinction between positive and negative association is unreliable for small absolute values of the test statistic. Under the null hypothesis, the one-sided **log.likelihood** statistic approximates a normal distribution (as the signed root of a chi-squared distribution with one degree of freedom).

Poisson.Stirling

Approximation of the likelihood of the observed cooccurrence frequency *O11* under the point null hypothesis of independence (so that the expected frequency is *E11*). The measure is derived from **Poisson.likelihood** (in the UCS::AM::HTest module) using Stirling's formula, resulting in a simple expression that can easily be evaluated. This measure was proposed by Quasthoff and Wolff (2002) and has been re-scaled to base 10 logarithms to allow a direct comparison with **Poisson.likelihood**.

Poisson.pv

Significance (one-sided p-value) of an exact Poisson test for the observed cooccurrence frequency O11 compared to the expected frequency E11 under the point null hypothesis of independence. This test is based on a Poisson approximation of the correct binomial sampling distribution of O11. It is numerically and analytically much easier to handle than the binomial test.

Fisher.pv

Significance (one-sided p-value) of Fisher's exact test for independence of rows and columns in a 2x2 contingency table with fixed marginals. This test is widely accepted as the most appropriate independence test for contingency tables (cf. Yates 1984). Its use as an association measure was suggested by Pedersen (1996).

MI

Maximum-likelihood estimate of the base 10 logarithm of the *mu*-value, which is identical to pointwise mutual information between the events describing occurrences of a pair's components. Note that mutual information is measured in *decimal units* rather than the customary *bits*. The theoretical range is from *-inf* to *+inf*, but the actural range for a given data set is restricted depending on the sample size *N*.

MI2

A heuristic variant of **MI** where the numerator is squared in order to discount low-frequency pairs. This measure also has some theoretical justification, being the square of the **gmean** measure.

MI3

Another heuristic variant of **MI** where the numerator is cubed, which boosts the discounting effect considerably.

relative.risk

Maximum-likelihood estimate of the logarithmic relative risk coefficient of association strength (base 10 logarithm). Ranges from *-inf* to *+inf*.

odds.ratio

Maximum-likelihood estimate of the logarithmic odds ratio as a coefficient of association strength (base 10 logarithm). Ranges from *-inf* to *+inf*.

odds.ratio.disc

A "discounted" version of **odds.ratio**, adding 0.5 to each factor in the equation. This modification of the odds ratio is commonly used to avoid infinite values, but does not seem to have a theoretical foundation.

Dice

Maximum-likelihood estimate of the Dice coefficient of association strength. Ranges from 0 to 1.

Jaccard

Maximum-likelihood estimate of the Jaccard coefficient of association strength, which is equivalent to **Dice** (i.e., there is a strictly monotonic mapping between the two association scores). Ranges from 0 to 1.

MS

Maximum-likelihood estimate of the *minimum sensitivity* coefficient suggested by Pedersen and Bruce (1996). Ranges from 0 to 1.

gmean

Maximum-likelihood estimate of the *geometric mean* coefficient of association strength. Ranges from 0 to 1.

average.MI

Maximum-likelihood estimate of the average mutual information between the indicator variables X and Y marking instances of a pair type's components. This implementation uses base 10 logarithms and multiplies the mutual information value with the sample size N in order to obtain readable values. Interestingly, **average.MI** is identical to Dunning's log-likelihood measure (**log.likelihood** and its variants) except for a scaling factor.

local.MI

Contribution of a given pair type to the (maximum-likelihood estimate of the) average mutual information of *all* cooccurrences. Formally, this is the mutual information between the random variables U and V, which represent the component types of a pair token in the random sample.

ADD-ON PACKAGES

The **UCS::AM** module provides a basic set of useful and well-known association measures. Except for the **Poisson.pv** and **Fisher.pv**, all measures have simple equations that can be computed efficiently. Further and more specialised association measures can be imported from add-on packages. Currently, the following packages are available:

```
UCS::AM::HTest          variants of hypothesis tests, likelihood measures
UCS::AM::Parametric     parametric association measures
```

These packages are implemented as Perl modules and can simply be loaded with the **use** operator. Alternatively, the **UCS::Load_AM_Package** function provides a convenient interface, where only the last part of the package name has to be specified, is case-insensitive, and may be abbreviated to a unique prefix. For instance, the **UCS::AM::HTest** package can be loaded with the specification `'ht'`. The empty string `''` loads **UCS::AM**, and `'ALL'` imports all available AM packages. (See the UCS manpage for details.)

COPYRIGHT

3.10 UCS::AM::HTest

More association measures based on hypothesis tests

SYNOPSIS

```
use UCS;
use UCS::AM::HTest;

@htest_AMs = UCS::AM_Keys();

# z.score.pv
# z.score.corr.pv
# t.score.pv
# chi.squared.tt
# chi.squared.tt.pv
# chi.squared.corr.tt
# chi.squared.corr.tt.pv
# chi.squared.pv
# chi.squared.corr.pv
# log.likelihood.tt
# log.likelihood.tt.pv
# log.likelihood.pv
# binomial.pv
# multinomial.likelihood.pv
# hypergeometric.likelihood.pv
# binomial.likelihood.pv
# Poisson.likelihood.pv
# Poisson.likelihood.Perl.pv
```

DESCRIPTION

This module contains some further **association measures** based on statistical hypothesis tests, most of which are variants of measures defined in the **UCS::AM** module. There are also several likelihood measures, which compute the probability of the observed contingency table rather than applying a full hypothesis test. The association measures defined in this module are intended mainly for a detailed comparative study of the properties of the significance-of-association class of AMs. Casual users should stick with the variants found in the **UCS::AM** module.

The following section gives a full listing of the association measures defined in the **UCS::AM::HTest** module with short explanations. Please refer to *http://www.collocations.de/AM/* for the full equations and references. When the module is imported, the additional measures are registered with the **UCS** core library (see the UCS manpage for details on how to access registered association measures).

The association scores of measures with the suffix **.pv** can be interpreted as probabilities (i.e. the likelihood of the observed data or the p-value of a statistical hypothesis test). Such probabilities are given as **negative base 10 logarithms**, ranging from 0 to *+inf* (*+inf* is represented by the return value of the built-in **inf** function (see the UCS::Expression::Func manpage). Measures with the suffix **.tt** (for *two-tailed*) are derived from two-sided statistical hypothesis tests. One-sided versions of these tests are provided under the same name without the suffix.

ASSOCIATION MEASURES

z.score.pv

> The significance (one-sided p-value) corresponding to **z.score**, obtained from the distribution function of the standard normal distribution. (The **z.score** measure computes a z-score for the observed cooccurrence frequency O11 compared to the expected frequency E11; see the UCS::AM manpage for details.)

z.score.corr.pv

The significance (one-sided p-value) corresponding to **z.score.corr**, a z-score for O11 against E11 with Yates' continuity correction applied.

t.score.pv

The significance (one-sided p-value) corresponding to **t.score**, obtained from the distribution function of the standard normal distribution. Since the number of degrees of freedom is very large, the t-distribution of the test statistic is practically identical to the standard normal distribution (t-distribution with *df=inf*). (The **t.score** measure is an application of Student's t-test to the comparison of O11 against E11; see the UCS::AM manpage for details.)

chi.squared.tt

Pearson's chi-squared test for independence of rows and columns in a 2x2 contingency table. The equation used in this implementation is derived from the homogeneity version of the chi-squared test (for equality of the success probabilities of two independent binomial distributions), and is fully equivalent to that of the independence test. Note that Pearson's chi-squared test is two-sided.

chi.squared.tt.pv

The significance (two-sided p-value) corresponding to **chi.squared.tt**, obtained from the chi-squared distribution with one degree of freedom.

chi.squared.corr.tt

Pearson's chi-squared test for independence of rows and columns in a 2x2 contingency table, with Yates' continuity correction applied (two-sided test).

chi.squared.corr.tt.pv

The significance (two-sided p-value) corresponding to **chi.squared.corr.tt**.

chi.squared.pv

The significance (one-sided p-value) corresponding to **chi.squared**, the one-sided version of Pearson's test for the independence of rows and columns (see the UCS::AM manpage for details). The p-value is obtained from the standard normal distribution (since the signed square root of the chi-squared test statistic has a standard normal distribution).

chi.squared.corr.pv

The significance (one-sided p-value) corresponding to **chi.squared.corr**, the one-sided version of Pearson's chi-squared test with Yates' continuity correction applied. Again, the p-value is obtained from the standard normal distribution.

log.likelihood.tt

The log-likelihood statistic suggested by Dunning (1993), a likelihood ratio test for independence of rows and columns in a 2x2 contingency table. (Dunning introduced the statistic as a test for homogeneity of the table columns, i.e. equal success probabilites of two independent binomial distributions). Note that all likelihood ratio tests are two-sided tests.

log.likelihood.tt.pv

The significance (two-sided p-value) corresponding to **log.likelihood.tt**, obtained from the chi-squared distribution with one degree of freedom.

log.likelihood.pv

The significance (one-sided p-value) corresponding to **log.likelihood**, the one-sided version of Dunning's likelihood ratio test (see the UCS::AM manpage for details). The p-value is obtained from the standard normal distribution (since the signed square root of the log-likelihood statistic has a standard normal distribution.)

binomial.pv

Significance (one-sided p-value) of an exact binomial test for the observed cooccurrence frequency O11 compared to the expected frequency E11 under the point null hypothesis of independence. This test is computationally expensive and may be numerically unstable, so use with caution. (This is also the reason why it is not included in the **UCS::AM** module.)

multinomial.likelihood.pv

Likelihood of the observed contingency table under the point null hypothesis of independence (i.e. with expected frequencies E11, E12, E21, and E22 estimated from the observed table).

hypergeometric.likelihood.pv

Likelihood of the observed contingency table under the null hypothesis of independence of rows and columns, with all marginal frequencies fixed to the observed values.

binomial.likelihood.pv

Binomial likelihood of the observed cooccurrence frequency O11 under the point null hypothesis (with expected frequency E11 estimated from the observed table). This function is relatively slow and may be numerically unstable, so use with caution.

Poisson.likelihood.pv

Poisson approximation of the binomial likelihood **binomial.likelihood.pv**, which is numerically and analytically more manageable.

Poisson.likelihood.Perl.pv

Alternative version of **binomial.likelihood.pv**, based on a direct Perl implementation of the naive multiplicative algorithm.

COPYRIGHT

3.11 UCS::AM::Parametric

Parametric association measures

SYNOPSIS

```
use UCS;
use UCS::AM::Parametric;

@parametric_AMs = UCS::AM_Keys();

# MI.conf
# MI.conf.<n>        [<n> = 2, 3, 5, 10, 50, 100, 1000]
# Poisson.mu.pv
# Poisson.mu.<n>.pv  [<n> = 2, 3, 5, 10, 50, 100, 1000, 10000]
```

DESCRIPTION

This module contains some parametric **association measures**, which are parametrised extensions of measures defined in the basic **UCS::AM** module. Parametric measures are a recent development in cooccurrence statistics, and the choice of appropriate parameter values is still very much a research question. Parametric measures will often be computationally expensive and may be numerically unstable, so novice users are advised to use the basic measures from the **UCS::AM** module instead.

The following section gives a full listing of the parametric association measures defined in the **UCS::AM::Parametric** module with short explanations. Please refer to *http://www.collocations.de/AM/* for the full equations and references. When the module is imported, the additional measures are registered with the **UCS** core library (see the UCS manpage for details on how to access registered association measures).

The association scores of measures with the suffix **.pv** can be interpreted as probabilities (i.e. the likelihood of the observed data or the p-value of a statistical hypothesis test). Such probabilities are given as **negative base 10 logarithms**, ranging from 0 to *+inf* (*+inf* is represented by the return value of the built-in **inf** function (see the UCS::Expression::Func manpage).

ASSOCIATION MEASURES

MI.conf

Conservative estimate for the base 10 logarithm of the *mu*-value (whose maximum-likelihood estimate is given by the **MI** measure). The association score computed by **MI.conf** is the lower endpoint of a two-sided confidence interval for *mu* at significance level **alpha**, which is specified by the `ALPHA` parameter (as a negative base 10 logarithm). The "usual" significance levels *.01* and *.001* correspond to `ALPHA=2` and `ALPHA=3`, respectively.

Please duplicate the **UCS::Expression** object returned by `UCS::AM_Expression("MI.conf")` before modifying the `ALPHA` parameter.

MI.conf.*ALPHA*

Versions of **MI.conf** with the `ALPHA` parameter pre-set to the value specified as part of the name. Available ***ALPHA*** values are **2**, **3**, **5**, **10**, **50**, **100**, and **1000**. For instance, **MI.conf.10** computes a two-sided confidence interval at significance level 1E-10.

Do not modify the `ALPHA` parameter of these association measures (in the **UCS::Expression** object returned by the **UCS::AM_Expression** function).

Poisson.mu.pv

Poisson test for *O11* under the modified point null hypothesis *pi = p1 * p2 * mu* (rather than the independence hypothesis *pi = p1 * p2* used by the **Poisson.pv** measure). The (non-logarithmic) value of *mu* is given by the `MU` parameter. For `MU=1`, the association scores computed by **Poisson.mu.pv** are identical to those of **Poisson.pv**.

Please duplicate the **UCS::Expression** object returned by `UCS::AM_Expression("Poisson.mu.pv")` before modifying the `MU` parameter.

Poisson.mu.*MU*.pv

Versions of **Poisson.mu.pv** with the `MU` parameter pre-set to the value specified as part of the name. Available ***MU*** values are **2**, **3**, **5**, **10**, **50**, **100**, **1000**, and **10000**.

Do not modify the `MU` parameter of these association measures (in the **UCS::Expression** object returned by the **UCS::AM_Expression** function).

COPYRIGHT

3.12 UCS::DS

Base class for data set implementations

SYNOPSIS

```
use UCS::DS;

$ds = new UCS::DS;                    # "virtual" data set
$ds->add_vars($name1, $name2, ...);   # append variables (= columns) in this order
$ds->delete_vars($name1, ...);        # delete variables (column 'gaps' are closed)

$type = $ds->var($name);              # check whether variable exists, returns data type
$index = $ds->var_index($name);       # column index of variable
@names = $ds->vars;                   # list all variables in column order

$ds->temporary($name, 1);             # mark variable as temporary (will not be saved)

@lines = $ds->comments;               # ordered list of comment lines
$ds->add_comments($line1, ...);       # append comment lines
$ds->delete_comments;                 # delete all comments
$ds->copy_comments($ds2);             # copy all comments from $ds2

@global_vars = $ds->globals;          # unordered list of global variable names
$value = $ds->global($var);           # return value of global variable $var
$ds->set_global($var, $value);        # set value of global variable (may be new variable)
$ds->delete_global($var);             # delete global variable
$ds->copy_globals($ds2);              # copy global variables from $ds2
```

DESCRIPTION

UCS::DS acts as a base class for **data set** managers (either file streams or in-memory representations). A **UCS::DS** object manages a list of **variables** (with names according to the UCS naming conventions detailed in *ucsfile*), and maps them to the **column indices** of a data set file.

It is always ensured that the column indices of a data set span a contiguous range starting at 0. New variables will be appended to the existing columns in the order of declaration. When a variables is deleted, all columns to its right are shifted to fill the gap.

When it is available, **UCS::DS** objects also store information from the **header** of a data set file. This information includes **comment lines** and **global variables** (see *ucsfile* for details).

METHODS

$ds = new UCS::DS;

Create a new **UCS::DS** object, with an empty list of variables. Normally, this constructor is only invoked implicitly by derived classes.

$ds = new UCS::DS $name1, $name2, ...;

Creates a **UCS::DS** object with the specified variables. Same as

```
$ds = new UCS::DS;
$ds->add_vars($name1, $name2, ...);
```

$ds->add_vars($name1, $name2, ...);

Add one or more variables *$name1*, *$name2*, ... to the data set. Variables that are already defined will be silently ignored. New variables are appended to the existing columns in the specfied order. *$name1*, *$name2*, ... must be valid UCS variable names.

$ds->delete_vars($name1, $name2, ...);

Delete the variables *$name1*, *$name2*, ... from the data set. Variables that are not defined in the data set will be silently ignored. When a variable has been deleted, all columns to its right are shifted to fill the gap. All arguments must be valid UCS variable names.

$type = $ds->var($name);

Check whether the variable *$name* is defined in the data set *$ds*. Returns the data type of the variable (`BOOL`, `INT`, `DOUBLE`, or `STRING`, see *ucsfile*), or **undef** if it does not exist.

$is_temp = $ds->temporary($name);

$ds->temporary($name, $val);

Mark variable *$name* as temporary (if *$val* is true) or permanent (if *$val* is false). The single-argument version returns true if the variable *$name* is temporary. Temporary variables are interpreted by in-memory representations of data sets. They may be deleted automatically and will not be written to data set files.

$index = $ds->var_index($name);

Get column index of variable *$name*. *$index* ranges from 0 to one less than the number of variables in the data set. Returns **undef** if the variable *$name* does not exist in the data set. It is recommended to test this condition with the **var** method first.

@names = $ds->vars;

Returns the names of all variables in this data set, sorted by their column indices. When saved to a data set file, the columns will appear in this order.

@lines = $ds->comments;

Returns all comment lines as an ordered list (i.e. as they would appear in a data set file). Comment lines are **chomp**ed and the initial `#` character (followed by an optional blank) is removed.

$ds->add_comments($line1, ...);

Add comment lines (which will be appended to existing comments). Like the data returned by the **comments** method, *$line1* etc. should not begin with a `#` character or end in a newline.

$ds->delete_comments;

Deletes all comment lines.

$ds->copy_comments($ds2);

Copies all comment lines from *$ds2*, which must be an object derived from **UCS::DS**. Existing comments of *$ds* are overwritten. This command is equivalent to

```
$ds->delete_comments;
$ds->add_comments($ds2->comments);
```

@global_vars = $ds->globals;

Returns the names of all global variables in alphabetical order. **NB:** global variable names must be valid UCS identifiers.

$value = $ds->global($var);

Returns the value of a global variable *$var* as a character string. If the global variable *$var* does not exist, returns **undef**.

$ds->set_global($var, $value);

Set global variable *$var* to the string *$value*. If *$var* does not exist, it is automatically added to the data set.

$ds->delete_global($var);

Delete a global variable. If *$var* does not exist, the method call will be silently ignored.

$ds->copy_globals($ds2);

Copies all global variables and their values from *$ds2*, which must be an object derived from **UCS::DS**. Any existing global variables off the data set *$ds* will be erased.

COPYRIGHT

3.13 UCS::DS::Stream

I/O streams for data set files

SYNOPSIS

```
use UCS::DS::Stream;

$ds = new UCS::DS::Stream::Read $filename;
die "format error" unless defined $ds;
# access variables, comments, and globals with UCS::DS methods
while ($ds->read) {
  die "read/format error"
    unless $ds->valid;                # valid row data available?
  $n = $ds->row;                      # row number
  $idx = $ds->var_index("am.logl");   # see 'ucsdoc UCS::DS'
  $logl = $ds->columns->[$idx];       # $ds->columns returns arrayref
  $logl = $ds->value("am.logl");      # short and safe, but slower
  $rowdata = $ds->data;               # returns hashref (varname => value)
  $logl = $rowdata->{"am.logl"};      # == $ds->value("am.logl")
}
ds->close;

$ds = new UCS::DS::Stream::Write $filename;
# set up variables, comments, and globals with UCS::DS methods
$ds->open;                            # write data set header
foreach $i (1 .. $N) {
  $ds->data("id"=>$i, "l1"=>$l1, ...);# takes hashref or list of pairs
  $ds->data("am.logl"=>$logl, ...);   # may be called repeatedly to add data
  $ds->columns($i, $l1, $l2, ...);    # complete list of column data
  $ds->write;                         # write row and clear data cache
}
$ds->close;
```

DESCRIPTION

UCS data set streams are used to read and write data set files one row at a time. When an **input stream** is created, the corresponding data set file is opened immediately and its header is read in. The header information can then be accessed through **UCS::DS** methods. Each **read** method call loads a single row from the data set file into an internal representation, from which it is available to the main program.

An **output stream** creates / overwrites its associated data set file only when the **open** method is called. This allows the main program to set up variables and header data with **UCS::DS** method calls. After opening the file, the data for each row is first stored in an internal representation, and then written to disk with the **write** method.

Note that there are no objects of class **UCS::DS::Stream**. Both input and output streams inherit directly from the **UCS::DS** class.

INPUT STREAMS

Input streams are implemented as **UCS::DS::Stream::Read** objects. When an input stream is created, the header of the associated data set file is read in. Header data and information about the variables in the data set can then be accessed using **UCS::DS** methods.

The actual data set table is then loaded one row (= pair type) at a time by calling the **read** method. The row data are extracted into an internal representation where they can be accessed with various methods (some of them being safe, others more efficient).

The **na** method controls whether missing values (represented by the string `NA` in the data set file) are recognised and stored internally as **undef**s, or whether they are silently translated into 0 (`BOOL`, `INT`, and `DOUBLE` variables) and the empty string (`STRING` variables), respectively.

$ds = new UCS::DS::Stream::Read $filename;

Open data set file *$filename* and read header information. Header variables and comments, as well as information about the variables in the data set can then be accessed with **UCS::DS** methods. If *$filename* is a plain filename or a partial path (i.e., neither a full relative or absolute path starting with `/` or `./` nor a command pipe) and the file is not found in the current working directory, the standard UCS libary is automatically searched for a data set with this name.

If there is a syntax error in the data set header, **undef** is returned. Note that the object constructor will **die** if the file *$filename* does not exist or cannot be opened for reading.

$ds->na(1);

Enables recognition of missing values represented by the string `NA` (as used by **R**). When enabled, missing values are represented by **undef**s. Otherwise, they will be silently translated into 0 (`BOOL`, `INT`, and `DOUBLE` variables) and the empty string (`STRING` variables), respectively. Use `$ds->na(0);` to disable missing value support, which is by default activated.

$ok = $ds->read;

Read one line of data from the data set file and extract the field values into an internal representation. Returns false when the entire data set has already been processed. Typically used in a **while** loop similar to the diamond operator: `while ($ds->read) {...}`.

$at_end = $ds->eof;

Returns true when the entire data set has been read, i.e. the logical complement of the value returned by the last **read** call.

$ok = $ds->valid;

Returns true if the internal representation contains valid row data. Currently, this only compares the number of columns in the file against the number of variables in the data set. Later on, values may also be syntax-checked and coerced into the correct data type.

$n = $ds->row;

Returns the current row number (of the row read in by the last **read** call, which is now stored in the internal representation).

$value = $ds->value($name);

Get value by variables name. Returns the value of variable *$name* currently stored in the internal representation. This method is convenient and safe (because it checks that the variable *$name* exists in the given data set), but incurs considerable overhead.

$cols = $ds->columns;

Return entire row data as an array reference. Individual variables have to be identified by their index, which can be obtained with the **var_index** method (`$cols->[$idx]`. Since index lookup can be moved out of the row processing loop, this access method is much more efficient than its alternatives. **NB:** the array *@$rowdata* is not reused for the next line of input and can safely be integrated into user-defined data structures.

$rowdata = $ds->data;

Returns hash reference containing entire row data indexed by variable names. Thus, the values of individual variables can be accessed with the expression `$rowdata->{$varname}`, similar to using the **value** method. Access with the **data** method is convenient for copying row data to an output stream. It is relatively slow, though, and should not be used in tight loops.

$ds->close;

Close the data set file. This method is automatically invoked when the object *$ds* is destroyed.

OUTPUT STREAMS

Output streams are implemented as **UCS::DS::Stream::Write** objects. After creating an output stream object, variables and header data are set up with the **UCS::DS** methods. The data set header is written to disk when the **open** method is called.

After that, the actual data set table is generated one row at a time. Row data is first stored in the internal presentation (using the **data** or the **columns** method), and then written to disk when the **write** method is called.

$ds = new UCS::DS::Stream::Write $filename;

Create output stream for data set file *$filename*. Note that this file will only be created or overwritten when the **open** method is called (in contrast to input streams, which open the data set file immediately).

$ds->open;

After setting up variables and header data (comment lines and global variables) with the respective **UCS::DS** methods, the **open** method opens the data set file and writes the data set header. If the file cannot be opened for writing, the **open** method will **die** with an error message.

$ds->data($v1 => $val1, $v2 => $val2, ...);

$ds->data($hashref);

Store data for the next row to be written in an internal representation. When using the **data** method, variables are identified by name (*$v1*, *$v2*, ...) and can be specified in any order. The variable-value pairs can also be passed with a single hash reference. Variables that do not exist in the data set will be silently ignored. The **data** method can be called repeatedly for a single row.

$ds->columns($val1, $val2, ...);

The **columns** method provides a more efficient way to specify row data. Here, all column values are passed in a single method call, and care has to be taken to list them in the correct order (namely, the order in which the variables were set up with the **add_vars** method). **NB:** the **data** and **columns** methods cannot be mixed. It is also not possible to set up the row data incrementally with repeated **columns** calls.

$ds->write;

Writes the row data currently stored in the internal buffer to the data set file, and resets the buffer (to **undef** values). Any **undef** values in the buffer (including the case where some variables were not specified with the **data** method) are interpreted as missing values and substituted by the string `NA`.

$ds->close;

Completes and closes the data set file.

EXAMPLES

The recommended way of **copying rows** from one data set file to another is to use the **data** methods of both streams, so that variables are copied by name rather than column position. It would be more efficient to pass row data directly (using the **columns** methods), but this approach is prone to lead to errors when the order of the columns is different between the input and output data sets.

The following example makes a copy of a data set file, adding an (enumerative) `id` variable if it is not present in the source file.

```
$in = new UCS::DS::Stream::Read $input_file;
die "$input_file: format error"
  unless defined $in;
@vars = $in->vars;
$add_id = not $in->var("id");

$out = new UCS::DS::Stream::Write $output_file;
$out->copy_comments($in);             # copy comments and
$out->copy_globals($in);              # global variables from input file
$out->add_vars("id")                  # conventionally, the "id" variables
  if $add_id;                         # is in the first column
$out->add_vars(@vars);
$out->open;                           # writes header to $output_file

while ($in->read) {
  die "read/format error"
    unless $in->valid;
  $out->data($in->data);              # copy row data by field name
  $out->data("id" => $in->row)        # use row number as ID value
    if $add_id;
  $out->write;
}

$in->close;
$out->close;
```

COPYRIGHT

3.14 UCS::DS::Memory

In-memory representation of data sets

SYNOPSIS

```
use UCS::DS::Memory;

$ds = new UCS::DS::Memory;            # empty data set
$ds = new UCS::DS::Memory $filename;  # read from file (using UCS::DS::Stream)

# access & edit variables, comments, and globals with UCS::DS methods

$pairs = $ds->size;                   # number of pair types
$ds->set_size($pairs);                # truncate or extend data set

$value = $ds->cell($var, $n);         # read entry from data set table
$ds->set_cell($var, $n, $value);      # set entry in data set table

$rowdata = $ds->row($n);              # returns hashref (varname => value)
$ds->set_row($n, $rowdata);           # set row data (ignores missing vars)
$ds->set_row($n, "f1"=>$f1, "f2"=>$f2, ...);
$ds->append_row($n, $rowdata);        # append row to data set
$ds->delete_rows($from, $to);         # delete a range of rows from the data set

$vector = $ds->column($var);          # reference to data vector of $var
$vector->[$n] = $value;               # fast direct access to cells

$ds->eval($var, $exp)                 # evaluate expression on data set & store in $var
  unless $ds->missing($exp);          # check first whether all reqd. variables are available
$ds->add($var);                       # auto-compute variable (derived variable or registered AM)

$stats = $ds->summary($var);          # statistical summary of numerical variable

$ds->where($idx, $exp);               # define index: rows matching UCS expression
$n = $ds->count($exp);                # number of rows matching expression
$vector = $ds->index($idx);           # returns reference to array of row numbers
$ds->make_index($idx, $row1, $row2, ...);  # define index: explicit list of row numbers
$ds->make_index($idx, $vector);           #            or array reference (will be duplicated)
$ds->activate_index($idx);            # activate index (will be used by most access methods)
$ds->activate_index();                # de-activate index
$ds->delete_index($idx);              # delete index

$ds2 = $ds->copy;                     # make physical copy of data set (using index if activated)
$ds2 = $ds->copy("*", "am.%");        # copy selected variables only (in specified order)

$ds->renumber;                        # renumber/add ID values as increasing sequence 1 .. size

$ds->sort($idx, $var1, $var2, ...);   # sort data set on $var1, breaking ties by $var2 etc.
$ds->sort($idx, "-$var1", "+$var2");  # - = descending, + = ascending (default depends on variable type)
$ds->rank($ranking, $key1, ...);      # compute ranking (with ties) and store in data set variable $ranking

$ds->save($filename);                 # save data set to file (using index if activated)

$dict = $ds->dict($var1, $var2, ...); # lookup hash for variable(s) (UCS::DS::Memory::Dict object)
($max, $average) = $dict->multiplicity; # maximum / average number of rows for each key
if ($dict->unique) { ... }            # whether every key identifies a unique row
@rows = $dict->lookup($x1, $x2, ...); # look up key in dictionary, returns all matching rows
$row  = $dict->lookup($x1, $x2, ...); # in scalar context, returns first matching row
@rows = $dict->lookup($other_ds, $n); # look up row $n from other data set
$n_rows = $dict->multiplicity($x1, $x2, ...);  # takes same arguments as lookup()
@keys = $dict->keys;                  # return unsorted list of keys entered in dictionary
```

DESCRIPTION

This module implements an in-memory representation of UCS data sets. When a data set file has been loaded into a **UCS::DS::Memory** object (or a new empty data set has been created), then **variable names**, **comments**, and **globals** can be accessed and modified with the respective **UCS::DS** methods (see the UCS::DS manpage).

Additional methods in the **UCS::DS::Memory** class allow the user to:

- read and write individual **cells** as well as entire **rows** or **columns**
- change the **size** of a data set
- annotate **derived variables**, **association scores**, or arbitrary **UCS expressions** in the data set

- compute statistical **summaries** of numerical variables
- **select** rows matching given UCS expression from a data set
- **sort** data sets by one or more variables and compute **rankings**
- save the data set into a data set file

The individual methods are detailed in the following sections. In all methods, columns are identified by the respective variable names, whereas rows (corresponding to pair types) are identified by row numbers. **NB:** Row numbers start with 1 (like R vectors, but unlike Perl arrays)!

GENERAL METHODS

$ds = new UCS::DS::Memory;

Create empty data set. The new data set has zero rows and no variables. Returns object of class **UCS::DS::Memory**;

$ds = new UCS::DS::Memory $file [, '-na' ;]

Reads data set file into memory and returns **UCS::DS::Memory** object. The argument *$file* is either a string giving the name of the data set file or a **UCS::DS::Stream::Read** object (see the UCS::DS::Stream manpage), which has been opened but not read from. When the specified file does not exist and in the case of a read error, the constructor **die**s with an appropriate error message.

The option `'-na'` disables missing value support (which is enabled by default), so that `NA` values in the data set file will be replaced by 0 or the empty string, depending on the data type. Use `'+na'` to enable missing value support explicitly.

$V = $ds->size;

Returns the size of the data set, i.e. the number of rows (or pair types).

$ds->set_size($V);

Change the size of the data set to *$V* rows. This method can both truncate and extend a data set. **NB:** Unlike the **size** method, **set_size** always applies to the real size of the data set and ignores the active row index. However, all row indices are preserved and adjusted in case of a truncation. If there is an active row index, it remains active. (See the section §3.14 below for more information on row indices.)

$value = $ds->cell($var, $n);

Retrieve the value of variable *$var* for row *$n* (i.e. the *$n*-th pair type). This method is convenient and performs various error checks, but it involves a considerable amount of overhead. Consider the **column** method when performance is an issue.

$ds->set_cell($var, $n, $value);

Set the value of variable *$var* for row *$n* to *$value*. Like **cell**, this method is convenient, but comparatively slow. Consider the **column** method when is an issue.

$rowdata = $ds->row($n);

Returns hash reference containing the entire data from row *$n* indexed by variable names. This method is inefficient and mainly for convenience, e.g. when applying a UCS expression to individual rows (cf. the description of the **eval** method in the UCS::Expression manpage).

$ds->set_row($n, $rowdata);

$ds->set_row($n, $var1 => $val1, $var2 => $val2, ...);

Set the values of some or all variables for row *$n*. The values can either be passed in a single hash reference indexed by variable names, or as *$var* => *$value* pairs. Any variables that do not exist in the data set *$ds* are silently ignored. This method is faster than calling **set_cell** repeatedly, especially when a new row is added to the data set.

$ds->append_row($rowdata);

$ds->append_row($var1 => $val1, $var2 => $val2, ...);

Append new row to the data set and fill it with the specified values. This method is a combination of **set_size** and **set_row**. Variable values that are not specified in the argument list are set to **undef**. When there is an active row index, the new row is appended to this index, while all other indices remain unchanged (see the section §3.14 below for more information on row indices).

$ds->delete_rows($from, $to);

Delete rows *$from* through *$to* from the data set. **NB:** This method always applies to the real row numbers and ignores the active row index. All existing indices are adjusted (which is an expensive operation) and an active row index remains activated. (See the section §3.14 below for more information on row indices.)

$vector = $ds->column($var);

Returns an array reference to the data vector of variable *$var*. *$vector* can be used both for read and write access, so care has to be taken that the data set isn't accidentally modified (e.g. through side effects of a **map** or **grep** operation on *@$vector*). Of course, activating a row index has no effect, since the **column** method gives direct access to the internal data structures. (See the section §3.14 below for more information on row indices.)

@missing_vars = $ds->missing($exp);

Determines whether all variables required to evaluate the UCS expression *$exp* (an object of class **UCS::Expression**) are defined in the data set *$ds*. Returns an empty list if *$exp* can be evaluated, and the names of missing variables otherwise.

$ds->eval($var, $exp);

Evaluate the UCS expression *$exp* (an object of class **UCS::Expression**) on the data set *$ds*, and store its values in the variable *$var*. When *$var* is a new variable, it is automatically added to the data set; Otherwise, the previous values are overwritten. This operation is *much* faster than repeatedly evaluating *$exp* for each row. For convenience, *$exp* can also be specified as a source string, which will be compiled on the fly. **NB:** The **eval** method always operates on the entire data set, even when a row index is activated. (See the section §3.14 below for more information on row indices.)

$ds->add($var);

Add a new variable to the data set and auto-compute its values, or overwrite an existing variable. *$var* must be the name of a **derived variable** such as `E11` or an **association score** such as `am.t.score` (see the ucsfile manpage for details).

$stats = $ds->summary($var);

Computes a statistical summary of the numerical variable *$var* (a numerical variable is a variable of data type `INT` or `DOUBLE`). *$stats* is a hash reference representing a data structure with the following fields:

```
MIN     ...  minimum value
MAX     ...  maximum value
ABSMIN  ...  smallest non-zero absolute value
ABSMAX  ...  largest absolute value
SUM     ...  sum of all value
MEAN    ...  mean (= average)
MEDIAN  ...  median (= 50% quantile)
VAR     ...  empirical variance
SD      ...  empirical standard deviation (sq. root of variance)
STEP    ...  smallest non-zero difference between any two values
NA      ...  number of missing values (undef's)
```

Note that some of these fields may be **undef** if they have no meaningful value for the given data set.

$ds2 = $ds->copy;

$ds2 = $ds->copy(@variables);

Duplicates a data set, so that *$ds2* is completely independent from *$ds* (whereas `$ds2 = $ds;` would just give another handle on the same data set). Comments and globals are copied to *$ds2* as well. Optionally, a list of variable names and/or wildcard patterns (see the ucsexp manpage) can be specified. In this case, only the selected columns will be copied. **NB:** If there is an active row index, the copy will only include the rows selected by the index, and they will be arranged in the corresponding order. However, no row indices are copied to *$ds2*. (See the section §3.14 below for more information on row indices.)

$ds->renumber;

When rows have been deleted from a data set, or a copy has been made with an active row index, the values of the `id` variable are preserved (and can be used to match rows against the correspond entries in the original data set). When an independent numbering is desired, the **renumber** method can be used to re-compute the `id` values so that they form an uninterrupted sequence starting from 1. **NB:** The renumbering ignores an activated row index.

$ds->save($filename);

$ds->save($filename, @variables);

This method saves the contents of *$ds* to a UCS data set file *$filename*. When an optional list of variable names and/or wildcard patterns (see the ucsexp manpage) is specified, only the selected columns will be saved. **NB:** If there is an active row index, only the rows selected by the index will be written to *$filename*, and they will be arranged in the corresponding order. The row indices themselves cannot be stored in a data set file. (See the section §3.14 below for more information on row indices.) Also note that **temporary variables** will not be saved (see the UCS::DS manpage).

ROW INDEX METHODS

A **row index** is an array reference containing a list of row numbers (starting from 1, unlike Perl arrays). Row indices are used to select rows from an in-memory data set, or to represent a re-ordering of the rows (or both). They are usually created by the **where** and **sort** methods, but can also be constructed explicitly. An arbitrary number of named row indices can be stored in a **UCS::DS::Memory** object.

A row index can be **activated**, creating a "virtual" data set containing only the rows selected by the index, arranged in the corresponding order. Most **UCS::DS::Memory** methods will then operate on this virtual data set. All exceptions are marked clearly in this manpage. In particular, the **where** method selects a subset of the activated index, and **sort** can be used to reorder it. There can only be one active row index at a time. There is no way of localising the activation (so that a previously active index is restored at the end of a block), so it is highly recommended to use active indices only locally and de-activate them afterwards.

Index names must be valid UCS identifiers, i.e. they may only contain alphanumeric characters (`A-Z a-z 0-9`) and periods (`.`) (cf. VARIABLES in *ucsfile*). Note that index names beginning with a period are reserved for internal use.

$ds->make_index($idx, $row1, $row2, ...);

$ds->make_index($idx, $vector);

Construct row index from a list of row numbers or an array reference *$vector*, and store it under the name *$idx* in the data set *$ds*. In the second form, the anonymous array is duplicated, so the contents of *$vector* can be modified or destroyed without affecting the stored row index.

$vector = $ds->index($idx);

Retrieve row index by name. Returns an array reference to the internal data, so be careful not to modify the contents of *$vector* accidentally. In most cases, it is easier to activate *$idx* and use the normal access methods.

$ds->delete_index($idx);

Delete the row index named *$idx*. If it happens to be activated, it will automatically de-activated.

$ds->activate_index($idx);

Activate row index *$idx*. This will clear any previous activations. Note that this operation may change the effective size of the data set as returned by the **size** method (unless *$idx* is just a sort index).

$ds->activate_index();

Deactivate the currently active index, re-enabling direct access to the full data set in its original order.

$ds->where($idx, $exp);

Construct *$idx* selecting all rows for which the UCS expression *$exp* (given as a **UCS::Expression** object) evaluates to true (see the ucsexp manpage for an introduction to UCS expression, and the UCS::Expression manpage for compilation instructions). It is often convenient to compile *$exp* on the fly, especially when it is a simple condition, e.g.

```
$ds->where("high.freq", new UCS::Expression '%f% >= 10');
```

which can be shortened to

```
$ds->where("high.freq", '%f% >= 10');
```

The **where** method will automatically compile the source string passed as *$exp* into a **UCS::Expression** object. On-the-fly compilation involves only moderate overhead. When there is an active row index, **where** will select a subset of this index, preserving its ordering.

$n = $ds->count($exp);

Similar to **where**, this method only counts the number of rows matching the UCS expression *$exp*, without creating a named index. The condition *$exp* may be given either as a **UCS::Expression** object or as a source string, which is compiled on the fly. (Internally, the rows are collected in a temporary index, which is automatically deleted when the method call returns.)

$ds->sort($idx, $key1, $key2, ...);

Sort data set *$ds* by the specified sort keys. The data set is first sorted, by *$key1*. Ties are then broken by *$key2*, any remaining ties by *$key3*, etc. If there are any ties left when all sort keys have been used, their ordering is undefined (and depends on the implementation of the **sort** function in Perl). The resulting ordering is stored in a row index with the name *$idx*. When there is an active row index, **sort** will re-order the rows selected by this index.

Each **sort key** consists of a variable name, optionally preceded or followed by a `+` or `-` character to select ascending or descending sort order, respectively. The default order is *descending* for Boolean variables and association scores, and *ascending* for all other variables. The sort keys `'l1'` and `'l2'` sort in alphabetical order, while `'f-'` puts the most frequent pair types first.

In order to break remaining ties randomly, an appropriate additional sort key has to be specified. The usual choice are the association scores of the **random** measure (see the UCS::AM manpage). It may be necessary to compute this measure first, which can be conveniently done with the **add** method, as shown in the example below.

```
# order pair types by frequency (descending), breaking ties randomly
if (not $ds->var("am.random")) {
  $ds->add("am.random");
  $ds->temporary("am.random", 1);  # temporary, don't save to disk
}
$ds->sort("by.freq", "f-", "am.random");
```

$ds->rank($ranking, $key1, $key2, ...);

The **rank** method is similar to **sort**, but creates a ranking instead of a sort index. The ranking is stored in the integer variable *$ranking*. Note that tied rows are assigned the same rank, which is the lowest available rank (as in the Olympic Games) rather than the average of all ranks in the group (as is often done in statistics). All other remarks about the **sort** method apply equally well to the **rank** method, especially those concerning randomisation.

DICTIONARIES (LOOKUP HASHES)

A data set **dictionary** is a **hash** structure listing all the different values that a given variable assumes in the data set (or all the different value combinations of several variables). For each value (or value combination), which is called a **key** of the dictionary, the corresponding row numbers in the data set can be retrieved (called a **lookup** of the key). In the terminology of relational databases, such a dictionary is referred to as an **index**. Be careful not to confuse this notion with the **row index** described above, which is used for subsetting and/or reordering the rows of a data set.

A dictionary can be created for any variable (or combination of variables) with the **dict** method, and is returned in the form of a **UCS::DS::Memory::Dict** object. **NB:** This dictionary is only valid as long as the data set itself is not modified (which includes activation or deactivation of a row index). Unlike a database index, the dictionary is not updated automatically. It is therefore important to keep operations on the data set under strict control while a dictionary is in use. It is always possible to add, modify, and delete variables that are not included in the dictionary, though. For the same reason (as well as to save working memory), dictionaries should be deleted when they are no longer needed.

The main purpose of a dictionary is to **look up** keys and find the matching rows in the data set efficiently (the **ucs-join** program is an example of a typical application). It is often desirable to choose variables in such a way that every key identifies a unique row in the data set (for instance, the values of `l1` and `l2` identify a pair type, which should have only one entry in a data set). A dictionary with this property is called **unique**. Both unique and non-unique dictionaries are supported (unique dictionaries are represented in a memory-efficient fashion). Lookup and similar operations are implemented as methods of the **UCS::DS::Memory::Dict** object.

Although mainly intended for string values, dictionaries support all data types. Boolean variables will usually be of interest only in combination with other variables (possibly also Boolean ones), and dictionaries are rarely useful for floating-point values.

$dict = $ds->dict($var1, ..., $varN);

Create a **dictionary** for the variables *$var1*, ..., *$varN* in the data set *$ds*. Each **key** of this dictionary is a combination of N values, which must be specified in the same order as the variable names. When a row index is in effect, keys and row numbers in the dictionary are taken from the virtual data set defined by the activated index. The returned object of class **UCS::DS::Memory::Dict** is a read-only dictionary: in order to take changes in the data set *$ds* into account (including the activation or deactivation of a row index), a new object has to be created with the **dict** method.

if ($dict->unique) { ... }

This method returns a true value iff *$dict* is a **unique** dictionary.

($max, $avg) = $dict->multiplicity;

$max = $dict->multiplicity;

Returns the maximum (*$max*) and average (*$avg*) number of rows matching a key in *$dict*. The dictionary is unique iff *$max* equals 1.

@rows = $dict->lookup($x1, ..., $xN);

$row = $dict->lookup($x1, ..., $xN);

Look up a **key**, specified as an *N*-tuple of variable values *($x1, ..., $xN)*, in the dictionary *$dict* and return the matching row numbers. The values *$x1*, ..., *$xN* must be given in the same order as the variables *$var1*, ..., *$varN* in the **dict** method call when the dictionary was created. When the key is not found in *$dict*, an empty list is returned.

In scalar context, the (number of the) first matching row is returned, or **undef** if the key is not found in the dictionary.

@rows = $dict->lookup($ds2, $n);

$row = $dict->lookup($ds2, $n);

The **lookup** method can also be used to look up rows from a second data set *$ds2*, i.e. to find rows in the dictionary's data set *$ds* where the values of *$var1*, ..., *$varN* match the *$n*-th row of *$ds2*. For this form of invocation, the dictionary variables must be defined in *$ds2* (otherwise, a fatal error is raised).

$n_rows = $dict->multiplicity($x1, ..., $xN);

$n_rows = $dict->multiplicity($ds2, $n);

When called with arguments, the **multiplicity** method returns the number of rows matching a specific key in *$dict*. The key can be given in the same two ways as for the **lookup** method. (Note that calling **lookup** in scalar context returns the first matching row, *not* the total number of rows.)

@keys = $dict->keys;

$n_keys = $dict->keys;

Returns an unsorted list of all dictionary keys in the internal representation (where each key is a single string value). Such internal representations can be passed to the **lookup** and **multiplicity** methods instead of an *N*-tuple *($x1, ..., $xN)*. In scalar context, the **keys** method efficiently computes the number of keys in *$dict*.

Examples The **keys** method and the ability to use the returned internal representations in the **lookup** method provide an easy way to compute the (empirical) **distribution** of a data set variable, i.e. a list of different values and their multiplicities. (Note that calling **lookup** in scalar context cannot be used to determine the multiplicity of a key because it returns the first matching row in this case.)

```
# frequency table for variable $v on data set $ds
$dict = $ds->dict($v);
@distribution =
  # sort values by multiplicity
  sort { $b->[1] <=> $a->[1] or $a->[0] cmp $b->[0] }
  # compute multiplicity for each value
  map { [$_, $dict->multiplicity($_)] }
  # for a single variable $v, internal keys are simply the values
  $dict->keys;
undef $dict;                  # always erase dictionary after use
```

The following example is a bare-bones version of the **ucs-join** command, annotating the pair types of a data set *$ds1* with a variable *$var* from another data set *$ds2* (matching rows according to the pair types they represent, i.e. using the variables `l1` and `l2`). Typically, *$ds2* will be an annotation database.

```
$ds1->add_variables($var);    # assuming $var hasn't previously exist in $ds1
$dict = $ds2->dict($var);
$dict->unique
  or die "Not unique -- can't look up pair types.";
foreach $n (1 .. $ds1->size) {
  $row = $dict->lookup($ds1, $n);
  $ds1->set_cell($var, $n, $ds2->cell($var, $row))
    if defined $row;
}
undef $dict;
```

SEE ALSO

The ucsfile manpage for general information about UCS data sets and the data set file format, the ucsexp manpage for an introduction to UCS expressions (which are used extensively in the **UCS::DS::Memory** module) and wildcard patterns, the UCS::Expression manpage for information on how to compile UCS expressions, and the UCS::DS manpage for methods that manipulate the layout of a data set and its header information.

COPYRIGHT

3.15 UCS::DS::Format

ASCII-format data set or subset

SYNOPSIS

```
use UCS::DS::Memory;
use UCS::DS::Format;

$ds = new UCS::DS::Memory $filename;  # needs in-memory representation

$formatter = new UCS::DS::Format $ds; # formatter object for data set $ds

$formatter->digits(6);                # number of significant digits

$formatter->mode("table");            # only mode so far

$formatter->pagelength(50);           # print in pages of 50 rows each
$formatter->pagelength(undef);        # print as single table

$formatter->vars($pattern, ...);      # select variables that will be shown

$formatter->print;                    # print formatted table on STDOUT
$formatter->print($filename);         # write to file or pipe
```

DESCRIPTION

This module provides a convenient method to format data sets as ASCII tables, which can then be used for viewing and printing. The formatter has to be applied to the in-memory representation implemented by the **UCS::DS::Memory** module. Its output is printed on STDOUT by default, but it can also be redirected to a file or pipe.

METHODS

$formatter = new UCS::DS::Format $ds;

Creates new formatter object for the data set *$ds*, which must be a **UCS::DS::Memory** object. The formatter object should be used immediately after its creation and destroyed afterwards. When any changes are made in the data set *$ds*, a new formatter has to be created.

$formatter->digits($n);

Configure *$formatter* to display approximately *$n* significant digits for floating-point variables (data type `DOUBLE`). *$n* must be at least 2.

$formatter->mode("table");

The default mode `table` prints the data set in the form of a simple ASCII table with column headers. It is the only supported mode so far.

$formatter->pagelength($rows);

Configure *$formatter* to format data set in separate pages of *$n* rows each. The individual pages are separated by a single blank line. Use of this option may improve the formatting quality, helps to avoid excessive columns widths, and reduces the delay before partial results can be displayed.

When *$rows* is set to 0 or omitted, the entire data set is printed as a single table. This is also the default behaviour.

$formatter->vars($pattern, ...);

Display only variables matching the specified wildcard patterns, in the specified order. This configuration option can also be used to change the ordering of the columns or display a

variable in more than one column. Repeated calls to the **vars** method will overwrite, rather than add to, the previous selection.

$formatter->print;

$formatter->print($filename);

Format the data set with the specified options, and print the result on STDOUT. When the optional argument *$filename* is specified, the output is redirected to this file or pipe.

SEE ALSO

See also the manpage of the PRINT utility, which is based on the **UCS::DS::Format** module.

COPYRIGHT

3.16 UCS::Mathlibs

Interface to optional mathematics libraries

SYNOPSIS

```
use UCS::Mathlibs;
```

DESCRIPTION

TODO ...

COPYRIGHT

Index