PetaMem Scripting Environment - Manual

PetaMem GmbH
Radlická
CZ-150 00 Prague
support@petamem.com

As of: October 12, 2025

Contents

List of Figures

List of Tables

Chapter 1

Deployment and Configuration

This chapter describes how to handle the installation(s) of PetaMem (PM) projects[1] on your server. That includes installation, deinstallation, upgrading, downgrading as well as information about the current installation layout.

The central tool for installation and deinstallation purposes is the `pm_install` script which you'll find shipped with every installation media.

```
$ ./pm_install -h
```

will output (version numbers may vary)[2] the usage information

```
This is the PetaMem 'pm_install' script rev. 747
 Copyright (C) PetaMem, s.r.o. 2008-present

 Usage: pm_install [options]
    -cpus                number of CPUs pm_install could use
                         (for performing testsuite)
    -debug               turns on debugging
    -delete <proj>=<rev> remove the revision <rev> for <proj>
    -disable <proj>      deactivate the currently active revision
    -help                prints this usage information and exits
    -list                lists all installed revisions for PetaMem projects
    -set <proj>=<rev>    set the revision <rev> for <proj> as the currently
                         active and exit
    -source=<path>       set the source directory manually
    -test                perform only a test of environment
```

1.1 Deinstalling of a PM Project

If you want to disable the currently active revision of PM project, the command

[1] There exist 4 PM projects now - namely: PMLS, PMSE, PMLIB, PMWF.

[2] Also please be aware, that the version of the `pm_install` script is not necessarily the same as the version of the PM project it will install on your server

```
$ pm_install --disable <proj>
```

will perform effectively a ”deinstallation” of the project from your server in a way that after applying this command, usually you won't be able to work with PMSE. Please be aware, that the software is not really deleted, but simply deactivated to prevent accidental data loss. If all you wanted was to switch between already installed versions, use the `-set <proj>=<rev>` option (see 1.2, pg. 2).

Use this command with caution, because after this PMLS will not start on your server until you again activate a version (either by installing a new one, or by issuing a `pm_install -set <proj>=<rev>` command. Also, keep in mind that by just deactivating a PMLS version, it still takes up space on your HD and with installing more and more versions you could run out of space on your mass storage.

!!! Removal of a PM project version !!!

As `-disable` only performs a deactivation of a PM project version, and this further takes up the space on your mass storage, you might at some point want to completely remove an obsolete version of the project from your server. The `-delete <proj>=<rev>` option is intended for this, as it will irrevocably remove the specified revision from disk. There are some security measures and side effects:

- The script will refuse to remove the newest revision installed. So if you intend to remove that, you have to do it manually from your OS shell.
- The user is presented with an interactive confirmation message before deletion.
- If the deleted version was the currently active one, the highest available version will be made active.

1.2 Upgrade and Downgrade

Upgrading and downgrading of a PM project is intuitive and straightforward.

1.2.1 By Installation

Any version you install is being made the currently active version. So if you install a newer version than what was before on your server, you have effectively upgraded a PM project. If you installed an older version than what was active on your server before, you have effectively downgraded a PM project.

Let's look at some examples. Assume you have done a fresh installation of PMLS, so you have just one version installed. This version is also the active one and your `/opt/PetaMem/PMSE` directory will look similar to this:

```
root@linux:/> ls -l /opt/PetaMem/PMSE
drwxr-xr-x  9 root root 4096 2015-09-20 16:31 home
lrwxrwxrwx  1 root root   13 2015-09-20 16:31 active -> rev-693
drwxr-xr-x  9 root root 4096 2015-09-20 16:31 rev-693
```

Assume you now obtain new installation media with a newer version. If you simply perform a new installation, the new version gets installed in parallel to the old one and becomes

automatically the active one. Your /opt/PetaMem/PMSE directory will then look similar to this:

```
root@linux:/> ls -l /opt/PetaMem/PMSE
drwxr-xr-x  9 root root 4096 2015-09-20 16:31 home
lrwxrwxrwx  1 root root   13 2015-09-20 16:31 active -> rev-719
drwxr-xr-x  9 root root 4096 2015-09-20 16:31 rev-693
drwxr-xr-x  9 root root 4096 2015-10-25 15:25 rev-719
```

What has happened here is, that the original installation (rev-593) has remained in place, the new installation (rev-619) has been installed in the PMSE_ROOT directory and has been marked as active by putting the ”active” link to it. So you can always inspect the active PMLS installation by changing to the directory.

The number of such parallel installations is only limited by the space on your hard disk. You probably have noticed the home directory. This contains the home directories of all named PMSE users and will neither change nor be removed when installing new versions or upgrading/downgrading PMSE.

Although it may be obvious, we should mention explicitely, that ex factory higher revision numbers correspond to newer versions of PMSE.

Note: PMSE project has NOT home directory.

1.2.2 By Explicit Switch

If you have already several versions of a PM project installed on your server and would like to switch from one to another, you can do so with the pm_install -set <proj>=<rev> (see 1.2.2, pg. 3) command, by simply providing the revision of the PM project version that is going to be the active one.

Let’s assume, that after some time you have several revisions of PMLS installed on your server and PMSE_ROOT looks something like this:

```
root@linux:/> ls -l /opt/PetaMem/PMSE
drwxr-xr-x  9 root root 4096 2015-09-20 16:31 home
lrwxrwxrwx  1 root root   13 2015-11-24 13:11 active -> rev-811
drwxr-xr-x  9 root root 4096 2015-09-20 16:31 rev-693
drwxr-xr-x  9 root root 4096 2015-10-25 15:25 rev-719
drwxr-xr-x  9 root root 4096 2015-11-24 13:11 rev-811
drwxr-xr-x  9 root root 4096 2013-12-21 14:16 rev-922
```

So there are four revisions, and the active one is 811. There is a newer one (922) and two older revisions (693 and 719). If you want now to upgrade from 811 to 922, you simply issue

```
$ pm_install -set PMLS=922
```

which will result in a new active link, effectively disabling rev-811 and enabling rev-922:

```
root@linux:/> ls -l /opt/PetaMem/PMSE
drwxr-xr-x  9 root root 4096 2015-09-20 16:31 home
lrwxrwxrwx  1 root root   13 2015-12-21 14:20 active -> rev-922
drwxr-xr-x  9 root root 4096 2015-09-20 16:31 rev-693
drwxr-xr-x  9 root root 4096 2015-10-25 15:25 rev-719
drwxr-xr-x  9 root root 4096 2015-11-24 13:11 rev-811
```

```
drwxr-xr-x  9 root root 4096 2015-12-21 14:16 rev-922
```

You do not have to look up the /opt/PetaMem/PMSE directory to find out what versions are installed and which one is active. Simply issue the command

```
$ ./pm_install -list
```

which will return either a message

```
System has no PMLS installation yet.
System has no PMSE installation yet.
System has no PMWF installation yet.
```

if there has been no PMLS, PMSE or PMWF installed on this system yet. Otherwise, there will be a list of versions installed and the active one (if any) will be marked with a star. For the above case the output would look like this:

```
Current PMSE installation layout:
  593
  619
  711
* 822
```

1.2.3 By Implicit Switch

The only situation when the currently active version of a PM project gets switched without explicit user interaction is when `pm_install` tries to maintain a sane (i.e. working) PM project infrastructure on your server. Such a case has been mentioned in the disable/deinstallation section (see 1.1, pg. 1), when deleting a PM project revision - that happened also to be the active revision - would render the system unusable.

`pm_install` then implicitely performs an upgrade by pointing the ”active” link to the newest PM project version installed on your server, thus implicitely switching versions.

Chapter 2

About the PetaMem Scripting Environment

The PMSE is a suite of programs and scripts that provide comprehensive functionality for processing large amounts of textual data - so called corpora.

In case you are familiar with the UNIX tools sed[1] and AWK[2], you will already know not only the alleged progenitors and inspiration for Perl, but also a large part of the concepts behind batch text processing.

In case you are not be familiar with these tools, never mind: The PMSE will provide you with powerful and concise text processing capabilities.

Let's say you have one (or more) of the following tasks to do:

- translate all of your intranet documentation from one or more source languages to several other target languages and periodically synchronize the translations.
- perform a grammar and spell check on thousands of documents in your company.
- retrieve specific information from internet sources, and to perform actions triggered by the semantic information retrieved (data mining/information retrieval)
- perform various conversions (format, encoding), apply filters, categorize texts etc. on textual data.

then PMSE is your tool and probably soon your friend.

2.1 Conceptual Overview

Image 2.1 on page 6 shows an abstract overview of the wide range of processes that can be achieved by PMSE. Starting at the text retrieval, the process-chain continues with statistical data-mining and ends with the display of information. Special position has the `P_ici` script, which can start parallel processing and make working with PMSE scripts highly effective.

[1] see http://en.wikipedia.org/wiki/Sed
[2] see http://en.wikipedia.org/wiki/AWK_programming_language

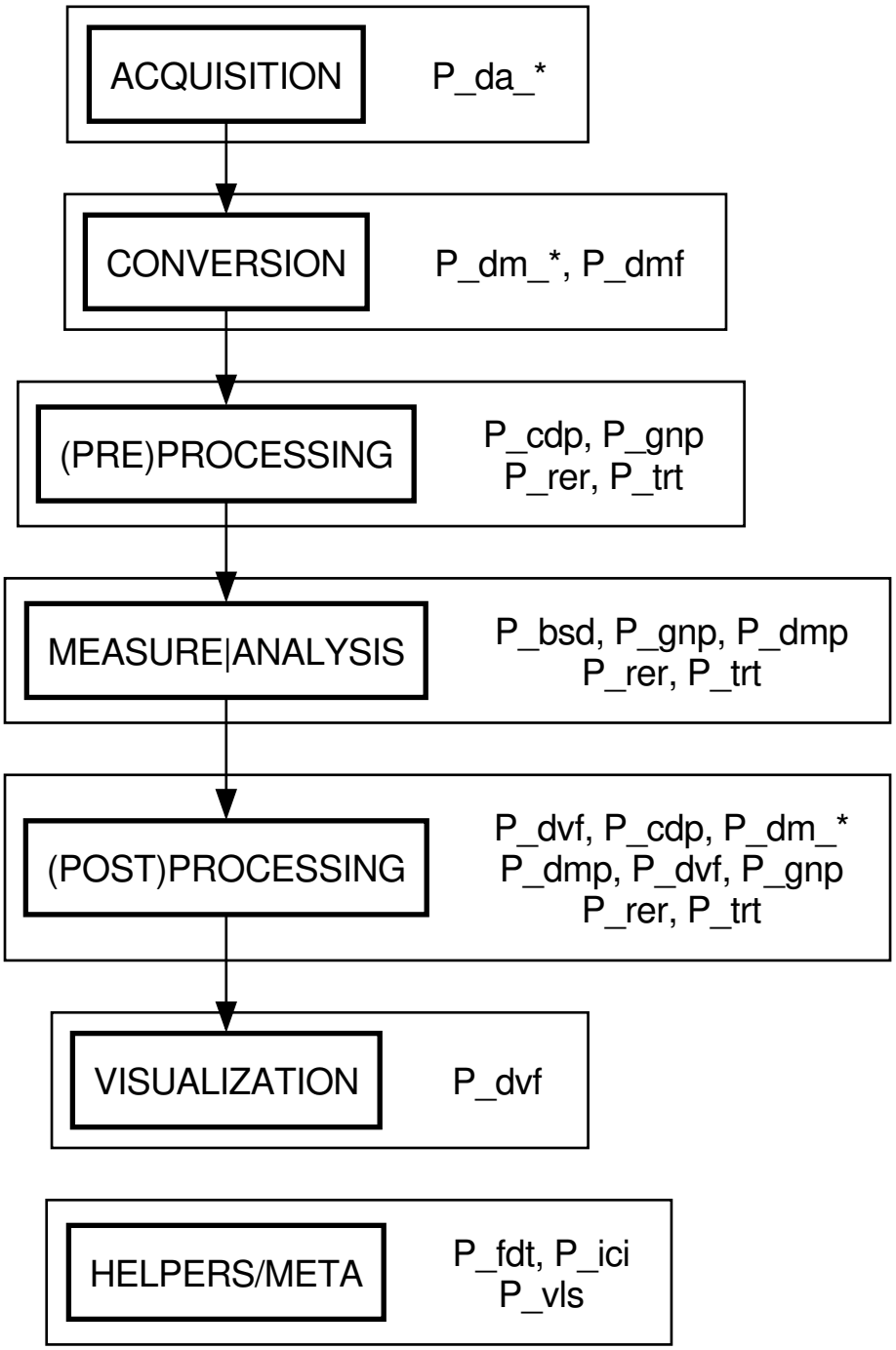

Figure 2.1: PMSE top level overview

2.2 Paths and Directory Structure

This section describes predeclared paths and presumed directory structure related to usage of PMSE. The paths are set in the environment by PetaMem configuration files.

2.2.1 PMSE - Root

The root of PMSE is set to be the `PMSE/active/` directory. It can be accessed in bash as:

```
$ cd $PMSE_ROOT
```

2.2.2 PMSE - Binary

The scripts described in the following section 2.3 are placed in `$PMSE_ROOT/bin` path which can be accessed from bash as a `$PMSE_BIN` variable.

```
$ cd $PMSE_BIN
```

2.2.3 Data - Library Structure

PMSE is designed to process texts in order to extract statistical data related to language. As we want PMSE to be a processor of multilingual data, we assume a big variability in sources of the data.

We adopted a specific name-space strategy to handle multilingual data and keep it synoptic. The resources are placed in `/data/library/` directory, which can be (in default setting) accessed in bash as

```
$ cd $PMCORP_ROOT
```

command. For information about conversion and manipulation of source files see P_dmf section 8.

2.3 PMSE Toolset Overview

Under the `PMSE_ROOT/bin` directory you will find several scripts that provide generic batch processing functionality and macros which alleviate the job of large scale text data processing. The naming scheme of the central PMSE scripts is P_xxx, where xxx is a mnemonic for a more verbatim explanation of the functionality of the specific command. In the same directory, a file `README.txt` will give you an overview of the scripts available: The most important scripts to become familiar with are the following:

P_bsd *Basic Statistical Data*
: Get basic statistical data about a corpus

P_cop *Co-occurrences Processor*
: Extract co-occurrences for given target from a list of bigrams

P_cct *Corpus Conversion Tool*
: Convert corpora and tagsets

P_csp *Comprehensive Statistics Processor*
: is the core statistical processing script in PMSE. It provides a wide range of statistical functions for text processing.

P_dmp *Distance Measures Processor*
: Computes distance measures for N-gram pairs

P_dmf *Data Mining Framework*
Process files in numerous formats into a plain text automatically

P_dvf *Data Visualization Framework*
Dump and convert data of various formats (Storable, YAML, ...)

P_fdt *File Distribution Tool*
Distribute given files to a set of containers of given size

P_gnp *Generic N-grams Processor*
N-grams processing, calculates various N-grams measurements, creating of contingency tables

P_help *PMSE Helper*
General help for PMSE environment

P_ici *Intelligent Command Iterator*
Script for iterating and parallelization of other scripts

P_rer *Regular Expression Replacer*
The regular expression replacement engine script

P_trt *Text Repair Tool*
Manipulate with a text on more abstract level

P_vls *Variable Length Splitter*
Specify a part of text (part of word list) and cut it out

Every script documents its commandline options via `<cmd> --help`. Also, as all of the scripts are an integral part of the PMSE infrastructure, they share several common options you will see in the help texts:

```
This is a PMSE script. Generic PMSE options are:
Options:
    --cpu <n>
        set the number of CPUs manually. This is recognized
        automatically and influences the number of parallel started
        processes (if possible). With this you can override the autodetect.

    --debug <n>
        Will enable the printout of debug messages. The verbosity threshold
        for the debug messages is handled in the same as for the regular
        messages. (see "--report").

    --dry
        dry run facility. Do not actually execute the commands

    --help
        print the script specific help and the general PMSE help

    --info
        information about the current environment, such as arguments,
        detected values, etc. When this parameter is detected the
        information delivered is current i.e. parameters AFTER this
        parameter are not considered yet. If you want the final
```

```
        information for all parameters, place this parameter last
        on the command line

    --report <n>
        by default all scripts have no output. If you'd like to know more
        about what's going on, set this option. Setting <n> higher will
        produce more verbose output.

    --version
        prints the version number of the script, and of PMSE, and exits.
```

Please note that command line parameter processing is done via the Perl module Getopt::Long[3] which implies some advanced and tolerant handling. For instance, it doesn't matter whether you provide `-h` or `-help` or `--help` as long as your option name is unambiguous.

There exists also option specific help available for options that require one or more input parameters from a list. This so called self-documenting help may be invoked with question-mark in the position of a argument for given option. In POD, it is marked as:

```
--option <type|?>

type may be one of following:

  A
  B
  C
  ...
```

All the most important scripts can be run in interactive mode when `--iact` option is used. In which case the user will be asked to complete options interactively. After execution, he can find the effective CLI in `pmse_env`.

The next sections will cover all PMSE scripts. The structure of the documentation is the same for every script: An introductory and general text, the command reference and examples.

While the examples are specific to the tool described, they often cover useful information about generic options and their usage, pitfalls, as well as tips and tricks. It is therefore advisable to look through all of them.

After this, the PMSE Tutorial and PMSE Cookbook chapters cover bigger use cases for PMSE.

[3] see http://search.cpan.org/perldoc?Getopt::Long

Chapter 3

P_bsd: Basic Statistical Data

`P_bsd` provides a basic statistical overview of a text/corpus. It is analogue to the Unix-like program *wc*, but it provides more options and more detailed output.

You can adjust the token definition by the `--delimiter` and `--ifilter` options. See section 6 for detailed usage of both of these options.

This script comes in handy, when you need to compare files without scanning them line by line. E.g.:The average number of characters per token or the type-token ratio could help you to detect strangeness in your text.

```
$ P_bsd -?
```

3.1 Reference

PMSE Basic Statistical Data Basic Statistical Data script gives an essential overview about the corpus.

USAGE P_bsd [options] file

At least one file (corpus) must be given.

The result is a Perl data structure stored in Storable format in `overview` file in directory specified by `--out` option.

OPTIONS

–delimiter <regex>

The delimiter has the form of Perl regular expression. It enables the tokenizer to dissect text into discrete tokens. If the user doesn't set his own value, the default from the PMLIB tokenizer is taken.

–ifilter [<type>=<regex>|?]*

This option may be provided multiple times (with different content for `type` of course) to define various filters, that are inserted at specific places in the data stream during processing. Valid values for ifilter <type> are:

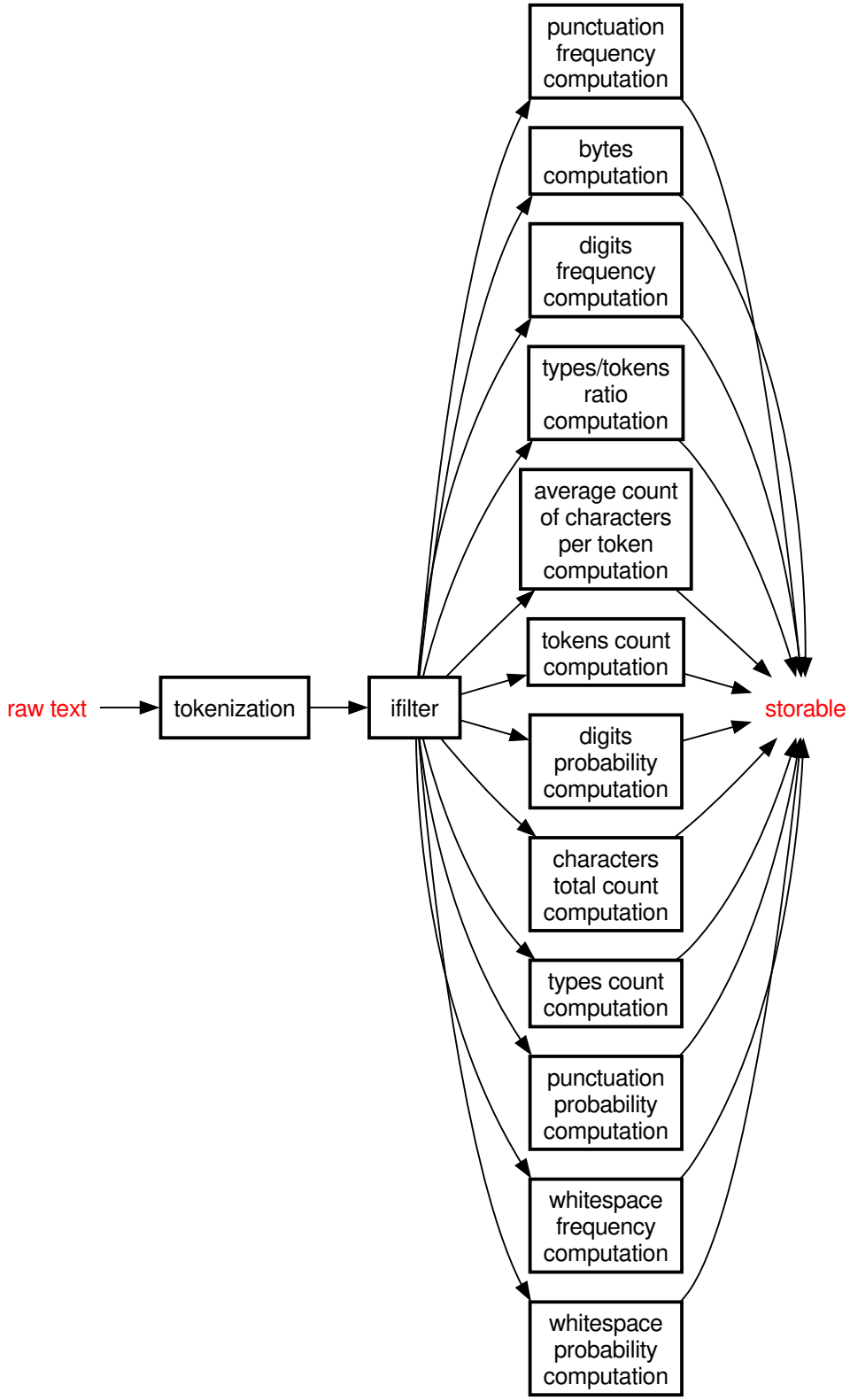

Figure 3.1: P_bsd schematic overview

```
+token    tokenizing step: matching tokens will pass
-token    tokenizing step: matching tokens will be blocked
```

<regex> may be any regular expression. Please take care to quote the whole expression, to prevent the shell modifying it. For example: '<filter>=<regex>'.

–**in** <**filename**>

Input file.

–**out** <**directoryname**>

Defines the output directory.

3.2 Examples

Basic information retrieval

```
$ P_bsd --out STDOUT --in text.txt
```

Tokenization will be provided by the default PMLIB tokenizer. The script will produce a hash structure with information about text specified in the `--in` option. This hash will be printed on STDOUT:

```
{
  'distribution_punctuation_freq' => 10055,
  'bytes'                         => 297858,
  'distribution_number_freq'      => 264,
  'types_tokens_ratio'            => '0.0479324517972524',
  'average_number_of_token_chars' => '2.55586541844361',
  'tokens_count'                  => 116539,
  'distribution_number_prob'      => '0.000886328384666519',
  'chars'                         => 297858,
  'types_count'                   => 5586,
  'distribution_punctuation_prob' => '0.0337576966205373',
  'distribution_whitespace_freq'  => 60847,
  'distribution_whitespace_prob'  => '0.204281906143196'
}
 (waiting for ENTER)
```

The abbreviations mentioned above mean:

- **average_number_of_token_chars** average number of characters per token
- **bytes** length in bytes
- **chars** length in characters
- **distribution_number_freq** frequency of digits in the text
- **distribution_number_prob** probability of digit occurrence
- **distribution_punctuation_freq** frequency of punctuation marks
- **distribution_punctuation_prob** probability of punctuation marks occurrence
- **distribution_whitespace_freq** frequency of whitespace
- **distribution_whitespace_prob** probability of whitespace occurrence
- **tokens_count** count of the tokens in the given text

- **types_count** count of the the types in the given text
- **types_tokens_ratio** type-token ratio

Chapter 4

P_cct: Corpus Conversion Tool

`P_cct` provides conversion of corpora from source format into Perl data structure. It also converts tagsets into Pmts - the PetaMem Tag Set. Conversion mechanism is implemented in PMSE::Corpus module (and all subsequent modules in given namespace). Conversion of tagsets takes place in PMLIB::Tag namespace. Architecture of corpus and tagset conversion is completely modular and may be extended easily.

```
$ P_cct -?
```

4.1 Reference

PMSE Corpus Converter Tool

USAGE P_cct [options]

P_cct converts corpus from original to specified form.

OPTIONS

–convert <**void**|**direction**|**?**>

Execute conversion. May be called without parameter, with specific direction and with `?`. If no parameter was specified, default conversion (original tagset 2 pmts) will be used. List of available conversions for given corpus may be listed with `?`.

–corpus <**corpus name**|**?**>

Specify name of corpus you want to convert. Currently are supported:

```
BNC
Penn
OANC
WikiCorpus
CNK
PDT
```

The complete list of corpora may be listed with `?` as a parameter.

–in <filename|directory>

Input files.

Also a directory may be specified, in that case specify also `mask`.

–mask <regex>

Positive filter for files. Default is:

```
\.txt\z
```

All files in input directory matching this regexp will be processed.

–out <directoryname> or <STDOUT>

Defines the output directory. If not specified, Data will be stored in CWD. Output file will have the same name as original file.

4.2 Examples

List available conversions

```
$ P_cct --corpus ?
```

Basic usage

```
$ P_cct --in OANC --out STDOUT --convert
```

Will find all files in OANC directory and will convert them into internal data structure. Penn-like tagset will be converted to Pmts.

Data conversion

Consider a conversion of OANC[1] corpus from previous example. Source files of OANC are stored in text format. A sample of such source is displayed below:[2]

```
"/'' But/CC you/PRP 're/VBP the/DT only/JJ
person/NN who/WP reads/VBZ it/PRP ./. "/'' That/DT may/MD be/VB
true/JJ ./. But/CC 15/CD ,/, 000/CD copies/NNS were/VBD printed/VBN ./.
```

Resulting data structure will look like this:

```
[
    [
         ["",     Oq   ],
         [That,   Dg--s],
         [may,    Voip ],
         [be,     Van  ],
         [true,   Afp  ],
         [.,      Ot   ],
     ],
```

[1] http://www.americannationalcorpus.org/

[2] Actually it is a segment of last paragraph of file *ArticleIP_2572.txt*.

```
    [
        [But,     Cc  ],
        [15,      Mc  ],
        [,,       Oy  ],
        [000,     Mc  ],
        [copies,  Nc-p],
        [were,    Vais],
        [printed, Vmps],
        [.,       Ot  ],
    ],
],
```

First level of braces encloses whole converted file, second encloses sentences and the lowest level holds 2-tuples - word and its tag.

Chapter 5

P_cop: Co-occurrence Processor

P_cop extracts co-occurrences for given target from a list of bigrams. The list must be stored in a Storable format. See section 18.5 for detailed instructions and explanation of the concept of co-occurrences.

Level of co-occurrences is specified via the `--level` option.

The target (a word with which co-occurs other words) may be specified as a regular expression or a literal. The output is an PMSE object, that can be easily visualized by P_dvf.

PMSE uses the GraphViz software[1] to display the relation between the target ant co-occurring words.

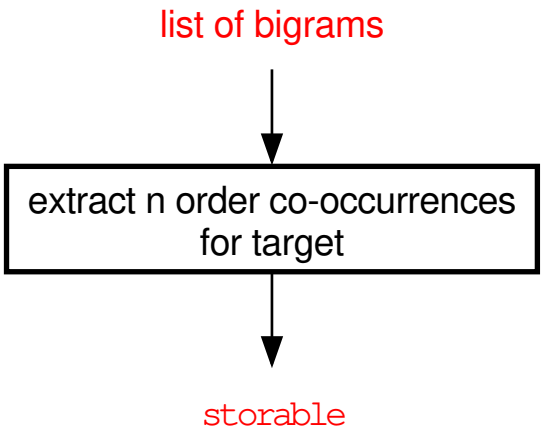

Figure 5.1: P_cop schematic overview

```
$ P_cop -?
```

5.1 Reference

PMSE Co-occurrence Processor

[1] http://www.graphviz.org/

USAGE P_cop [OPTIONS]

P_cop extracts co-occurrences from bigrams.

OPTIONS

–delimiter <**regexp**>

Defines delimiter of tokens in the n-gram. Default is white-space.

–in <**filename**>

Defines the input file.

It must be a Storable file. Input file must contain bigrams in the format:

```
<bigram> => <value>
```

–level <**n**>

Level of co-occurrences. Assume this example:

```
target:  A
 lvl 1:  A -> B (a co-occurrs with B)
 lvl 2:  B -> C
 lvl 3:  C -> D
```

Co-occurrence A -> D is of order 3. The ”deeper” level you go the more co-occurrences you get - and result may become messy. Filtering of input bigrams (and tokens) with high frequency may help here.

Default value of level is 3.

–out <**filename**>

Defines the name of the output file. The output is a PMSE::Visualize object stored in Storable format.

–query <**type**>=<**target**>|**?**

Type may be:

```
literal
regexp
```

It is necessary to specify a word / expression (target) for that we want to extract co-occurring words. The target may be specified as a literal string or as a regular expression. When a literal string is specified, co-occurrences will be extracted exactly for this string. Regular expression may match more words.

5.2 Examples

```
$ P_cop --in bigrams.sbl --out coocs --query 'regex=.+ship' --level 2
```

Chapter 6

P_csp: Comprehensive Statistics Processor

`P_csp` is a tool designed for extraction of tokens and their statistical characteristics, like basic count, frequency or probability. It is also capable to compute histograms of values for given set of tokens.

```
$ P_csp -?
```

6.1 Reference

PMSE Comprehensive Statistics Processor

USAGE P_csp [options]

P_csp is the core statistical processing script in PMSE. It provides a wide range of statistical functions for text processing.

The process workflow of P_csp is as follows:

```
text(s)-> tokenization -> statistics -> metrics -> output
```

Throughout this processing chain, various hooks are deployed and support extensive data mangling (filters, transformations etc.).

OPTIONS

–action [<**action**> | **?]***

You can give one or more named actions as a space-separated list to define what processing the script shall perform on the input files. Valid values for <action> are:

```
utcount    count unique tokens
utfreq     compute the token frequencies
utprob     compute the token probabilities
```

–bulk <**file**>

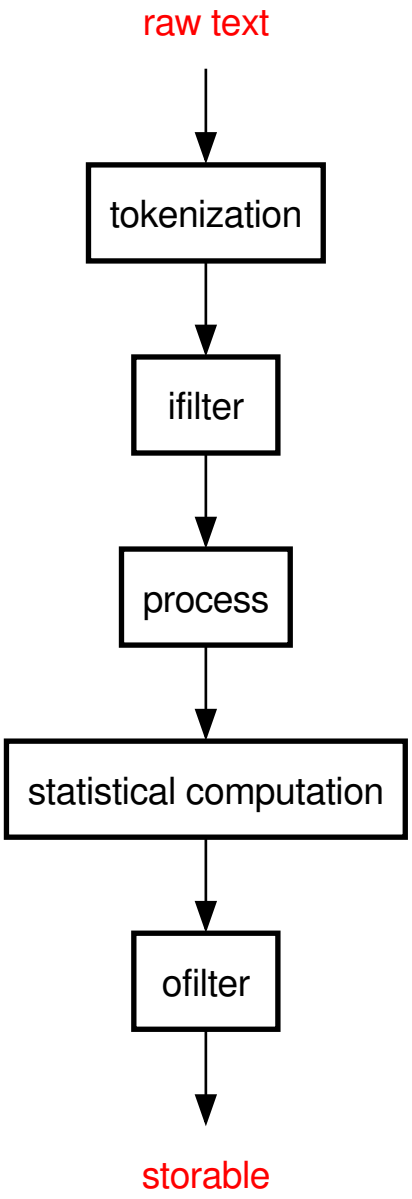

Figure 6.1: P_csp schematic overview

With option –bulk you can define an INI file, which can be used for histogram / hash filtering and tokens processing. The INI file has a structure of '[section]' and `name=value`, heredoc style can also be used. 'Section' is name of the procedure (process / ofilter), 'name' is the name of the hook (the same as in `--process` and `--ofilter`). 'Value' stores code, which will be executed on the result of the action.

–delimiter <**regex**>

Delimiter has the form of a Perl regular expression. It enables the tokenizer to dissect text into discrete tokens. If the user doesn't set his own value, the default delimiter from the PMLIB tokenizer is taken.

–histogram [<**action**>=<**file**> | **?]**

We can count a distribution (histogram) for the result of each 'action'. Histogram is stored in the file 'file'. Actions could be:

```
utcount   histogram stores distribution of tokens occurrences
utfreq    histogram stores distribution of tokens frequencies
utprob    histogram stores distribution of tokens probabilities
```

'File' is in Perl Storable format.

–ifilter [<**type**>=<**regex**> | **?]***

This option may be provided multiple times (with different content for `type` of course) to define various filters, that are inserted at specific places during data stream processing. Valid values for ifilter <type> are:

```
+token    tokenizing step: matching tokens will pass
-token    tokenizing step: matching tokens will be blocked
+utcount  unique tokens counting step: matching tokens will pass
-utcount  unique tokens counting step:
          matching tokens will be blocked
+utfreq   unique tokens frequencies step: matching tokens will pass
-utfreq   unique tokens frequencies step:
          matching tokens will be blocked
+utprob   unique tokens probabilities step: matching tokens will pass
-utprob   unique tokens probabilities step:
          matching tokens will be blocked
```

<regex> may be any regular expression. Please take care to quote the whole expression, like '<filter>=<regex>', to prevent the shell modifying it.

–in <**filename**>

Input file.

–ofilter [<**hook**>=<**code**> | **?]**

Ofilter can be used for result of each action, which is always a hash, thus with <code> we can affect both keys and values. Hook denotes the specific part of the process where the ofilter is applied. Hook can be:

```
utcount    filter result of action utcount
utfreq     filter result of action utfreq
utprob     filter result of action utprob
_hist      filter result before histogram is made
histogram  filter hash with histogram values
           (note: keys and values are both numeric)
```

Code could have the form of `$key =~ m{.*}xmsg` or `$value >=< number`. You can also insert the name of an INI file via the `--bulk` option.

–out <**directoryname**>

Defines the output directory.

–process [<**hook**>=<**subst**> | **?]***

<subst> is a Perl substitution operator: `s{pattern}{replacement}flags`, whereas "pattern" is a regular expression, "replacement" may be a string or even Perl code if the

”e” flag is given. Further info see e.g. http://perldoc.perl.org

<hook> may be one of the following:

```
_utcount      pre  'utcount' hook
 utcount_     post 'utcount' pre-set building
_utfreq       pre  'utfreq' set building
 utfreq_      post 'utfreq' set building
_utprob       pre  'utprob' set building
 utprob_      post 'utprob' set building
```

6.2 Examples

Basic usage

```
$ P_csp --action utcount --out tokens --in corpus.txt
```

will count all unique tokens (types) from the ”corpus.txt” file and the output will be saved in the directory called ”tokens”. The name of the output file is created from the combination of the `--action` function, or functions, which were called on the commandline. In this case, the path to the output file will be: ”tokens/utcount”.

To familiarize yourself with the dependencies of various processing options, please see figure 6.2.

Get frequencies

```
$ P_csp --action utfreq --ifilter '+token=\A\p{Alpha}+\p{Digit}* \z' \
> --insort alpha --out cfreq --in corpus.txt
```

will calculate the frequencies of alpha-numeric character tokens from the input file ”corpus.txt”. The contents of the output file will be sorted by ascending alphabetical order. The output, in this case, will be stored in the ”cfreq/utfreq” file.

Unique count, frequency, probability, lower casing

```
$ P_csp --action utcount utfreq utprob --ifilter '+token=\D+' \
> --delimiter '\s+' --process 's{\A(.+)\z}{lc($1)}xmse' \
> --out corpus_info --in corpus.txt
```

will count unique tokens, their frequency and probability. Tokens (in the original text) are split by whitespace. (`--delimiter`).

Only non-digits tokens will pass (`--ifilter`). All tokens will be lowercase (`--process`). The 3 output files: `utcount`, `utfreq`, `utprob` will be stored in the `corpus_info` directory.

The output of `P_csp` is a Perl data structure, further processing can be done by utilizing the `P_dvf` script.

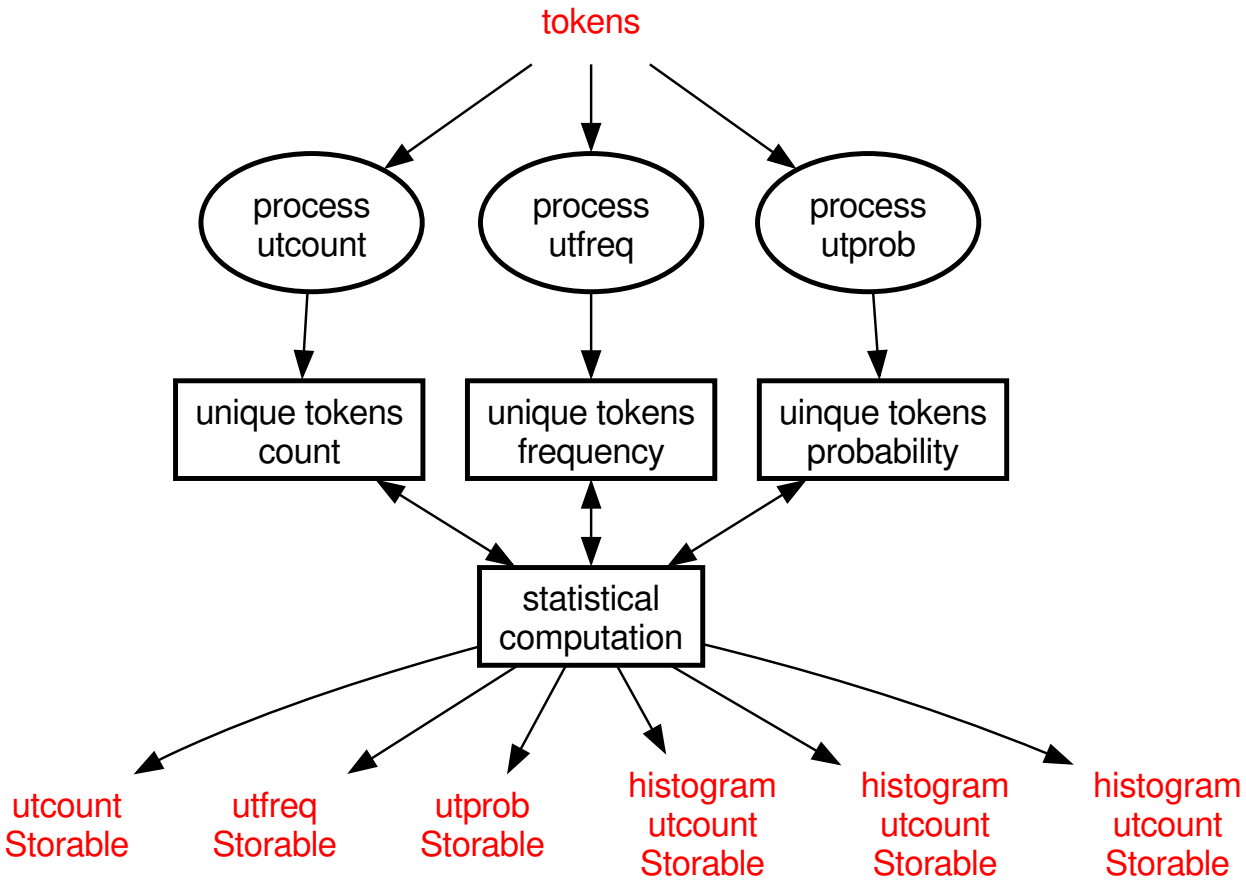

Figure 6.2: Overview of the statistical processes

How to create a bulk file?

The bulk file loaded into `P_csp` needs to comply with the INI file format: it must have the `[section]` and 'name=value' structure. Each closing heredoc terminator `END` must be followed by a newline.

```
[process]
_utcount=s{\d+}{NUMBER}xms
_utfreq=s{(A-Z)}{lc$1}xmse
utprob_=s{\bY.+\b}{}xmsi
[ofilter]
_ofilter=<<END
$key =~ m{\pP+}xmsg
$key !~ m{\w}xmsg && $value < 20
END
ofilter_=<<END
$value = 25|32|33
$value > 1500
END
```

Each '[section]' denotes the type of change, where 'name' is a hook, as in `--process` or `--ofilter` options. The 'value' contains code, that will be evaled during processing. You can use the heredoc style to include several lines of code and transformations. A bulk file could be used for multiple filtering and tokens processing.

Creating a histogram

```
$ P_csp --action utcount --out c_utcount \
> --histogram 'utcount=histogram' --in corpus.txt
```

Will calculate the histogram (distribution of occurrences for tokens), the histogram output will be stored in the file 'histogram'.

Creating a histogram and using ofilter

```
$ P_csp --action utcount --out c_utcount \
> --histogram 'utcount=histogram' --ofilter '_hist=$value > 20' --in corpus.txt
```

This is the same as the example above except that tokens which occur more than 20 times will be deleted. Deleted tokens won't be counted in the histogram. You can also use the `--bulk` option to apply multiple conditions, e.g.: you can delete various words, word forms and tokens represented by classes of regular expressions etc.

6.3 Q&A

Something is wrong while P_csp runs on commandline.

If you see this error, it is most probably caused by erroneous shell expansion. You will need to quote your regex `\b(\w).+\pP` or special characters `*`. Or you can escape them, by changing `*` to `\*`.

Pay especial attention when using the following characters: `[]<>()\*|`

Output file is not readable

The output of `P_csp` is saved in Storable format, which may need to be converted before being read by another script, perhaps using `P_dvf` or some other converter.

Chapter 7

P_daf: Data Acquisition Framework

P_daf is a framework for acquisition of various data. The rules (source, destination) for the acquisition are not dependent on P_daf. They are specified in a separate INI file.

That allows you to write your own INI files and make your data acquisition automated.

INI files are placed in a `$PMSE_ROOT/cfg/daf.d.` directory. But you can specify alternative path by the `--ini` option.

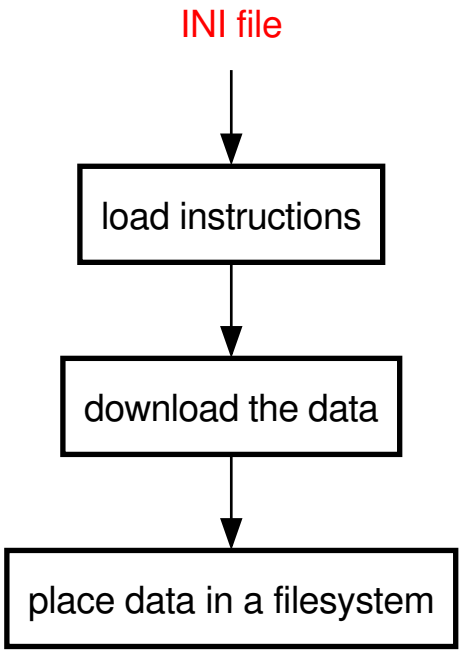

Figure 7.1: P_daf schematic overview

```
$ P_daf -?
```

7.1 Reference

PMSE Data Acquisition Framework

USAGE P_daf [OPTIONS]

P_daf downloads data from the web.

OPTIONS

–fetch [<PROJECT> | ?]

Specify project(s) to fetch. `?` will give you a list of available projects. If you store INI files in other than default directories, you should specify them in `--ini` option.

–ini [<DIR>,<FILE>]

Specify path(s) to INI file(s) or dir(s) which contain them. default is:

```
$PMSE_ROOT/cfg/daf.d
```

7.2 How to Write an INI File

First, let's take a look on a real one. The following INI file was used in PMSE to get list of geographic names:

```
[global]
lastfetch =
interval  = 6 months
name      = geonames

[geonames]
threads = 1
URL   = http://download.geonames.org/export/dump
match = (?<file>(allCountries|alternateNames).zip)
get   = "%URL%/$file"
store = "$ENV{PMCORP_ROOT}/m/u/l/original/geonames/$file"
```

The INI file has two sections: *global* and *geonames*.[1] Three parameters take place in the *global* section. *Lastfetch* may contain the date of the last download; this parameter is obligatory. *Interval* is the time period between downloads. *Name* is the name of the project, that you call from P_daf.

The section *geonames* contains several parameters:

threads specifies the number of processes

URL a web adress containing a hyperlink(s) to the data

match a regular expression that match the hyperlink(s) aiming on the data

get specifies the target (matched hyperlink) to download

store specifies the place in a filesystem, where to store the data

As you can see by the *get* parameter, templating can be used. The value of a previously defined parameter is accessed as a `%<name of parameter>%` template.

[1] The section name is always enclosed by square brackets.

The `$file` variable is specified as a named backreference (captured group). in *match* parameter.[2]

7.3 Private Data and Personal INI

Some of your projects may request login. In that case we recommend you to add 3 lines to the global section of the INI:

```
login    = 1      # this section will tell P_daf
                    that login info is required
password = XYZZ   # here you will store your password
username = NN     # and here username
```

Note: if you will specify `login = 1` and won't specify password or username, P_daf will skip your INI and give you some warning.

In the case you don't want to store your personal credentials in common directory, you may create a directory containing your personal INI in your home.[3] Even if there exists an INI of the same name in the common directory, P_daf will prefer yours. You have just to add your private directory to the `--ini` option:

```
# look for INIs in private and default directory
P_daf --fetch <private.ini> --ini <def.path> , <pers.path>
```

7.4 Extended INI File

The following INI file[4] shows how to fetch the newest data from a web site offering multiple targets sorted by a date. The base URL is an index of files to download.

```
[global]
lastfetch =
interval  = 6 months
name      = openstreetmap

[whole_planet]
BASE = http://ftp5.gwdg.de/pub/misc/openstreetmap/planet.openstreetmap.org/
       planet/2013/
url  = %BASE%
prehook_start =<<PREHOOK_START
    our $newest = 0;

    # 1. do we already have whole planet map? _NO_? then go on
    if (!glob("$ENV{PMCORP_ROOT}/m/u/l/original/openstreetmap/planet*")){
        # here we have to find the newest date
        while ($source =~ m{>(?<file>planet-(?<date>\d{6}).osm.bz2)<}xmsg){
            if ($+{date} > $newest){
```

[2] http://perldoc.perl.org/perlretut.html#Named-backreferences

[3] P_daf checks all paths to INIs and if there are duplicate INI names, it will prefer path containing your home address.

[4] The URL adress in the BASE (currently in 2 lines) line must be placed on one line. We broke the line because of readability.

```
                $newest = $+{date};
            }
        }
    }
PREHOOK_START
match   = \A   # exactly one match
get     = $newest ? "%BASE%/planet-$newest.osm.bz2" : undef
store   = "$ENV{PMCORP_ROOT}/m/u/l/original/openstreetmap/$file"
```

The `prehook_start` parameter contains a Perl code (in the heredoc format) which checks the input (BASE) file (web site) and searches for the latest date.

Hyperlinks aiming to the target files are matched with the

```
m{>(?<file>planet-(?<date>\d{6}).osm.bz2)<}xmsg
```

regexp. We have a named backreference called `$file`, which holds the name of each target. A date of origin is a part of the name. As we want to fetch only the newest file, we make a simply comparsion of all dates in the *while* loop. The latest date is assigned to the `$newest` variable.

The variable called `$source` holds the content of the BASE.

The *match* section contains a `\A` anchor, that cuases only one match on the given web site.

The *get* parameter contains a condition we can read as: ”do we have a defined variable `$newest` ? YES - fetch the newest file, NO - do nothing”.

7.5 Hooks

In previous section, we have mentioned `prehook_start` parameter. The concept of hook is to provide a ’raw’ Perl code that will be executed in a specific place (time) of the process of acquirement of the data. There are available four hooks, what makes the DAF config file pretty configurable. There exist 2 pre-hooks and 2 post-hooks. They are called:

```
prehook_start  # process BASE file -before spec. targets
prehook        # before a specific target is downloaded

posthook       # after a specific target is downloaded
posthook_final # after all downloads are completed
```

The whole process is described on figure 7.5. Phases of the data acqusition are visualized as box-like nodes. Red labels stand for hooks. *Get* and *store* parameters are similar to hooks, because their content may be modified by a Perl code.

With *get* you may modify the specific download link, with *store* the path, where do you want to store the data.

7.6 Examples

```
$ P_daf --fetch geonames
```

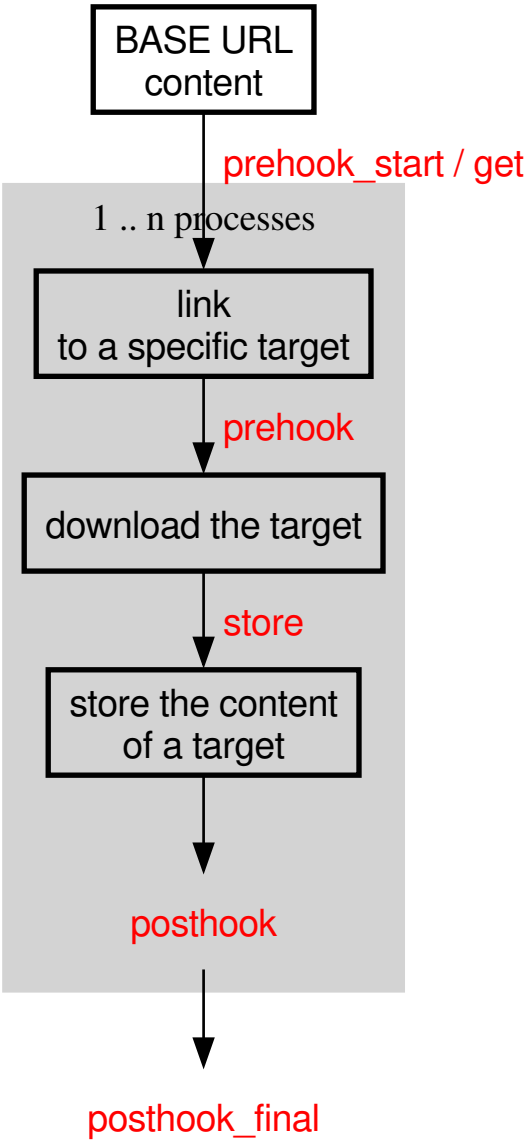

Figure 7.2: P_daf processes overview

Chapter 8

P_dmf: Data Mining Framework

The `P_dmf` tool allows to convert numerous input formats to plain text. It has also some extended features like Wikimedia projetcs processing. P_dmf is designed to work with specific file structure, which is described below.

```
$ P_dmf -?
```

8.1 Reference

SYNOPSIS P_dmf [options]

OPTIONS

–base <**base path**>

Base path of your corpus data. Default is '/data/library'.

–conv <**all|?**>

This option will list available converters. The resulting list differs according to input option:

```
all   all available modules in PMLIB
?     converters installed in the system
```

–in <**file|'glob'**>+

Defines the input source file or glob. Option can be given multiple times for multiple input source files. If you define a glob, please be aware to put it in single quotes to prevent shell expansion.

For this file you could also provide config file .dmf.info. You should put this file into one of parent dir. The closest one is applied.

–out <**dir**>

In case only one input dir is specified, P_dmf can store results into specified directory.

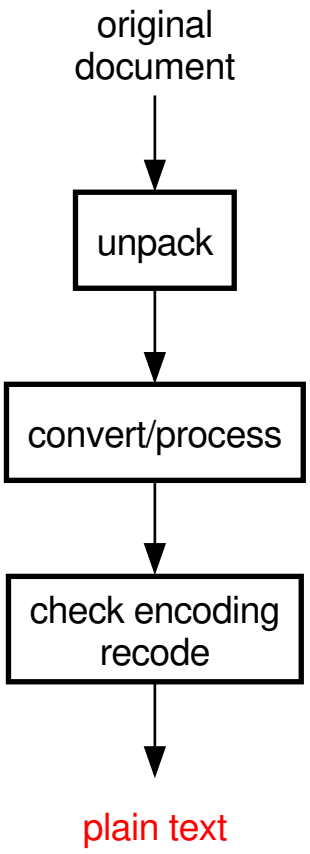

Figure 8.1: P_dmf schematic overview

8.2 File Structure

Consider a situation, when the amount of the textual documents is dynamically changing. In the filesystem exist old, yet processed documents and also come new - in various formats, encoding etc. PMSE adopted a specific strategy, how to handle this.

PMSE uses by default the path `/data/library/` as document root to store any processed documents. This can be changed by the environment variable `PMCORP_ROOT`.

The environment of PMSE is extensively multilingual, therefore it offers a very generic way to store data for numerous languages under this document root. Each language is represented as a directory trie[1]. The name of the trie is derived from the appropriate iso-639-3[2] code of the language.

The path for documents in English is therefore:

```
/data/library/e/n/g/
```

Original documents (incoming, raw data) are meant to be stored in a subdirectory called **original** and processed documents in a subdirectory **derived**. The resulting skeleton structure looks like this

[1] http://en.wikipedia.org/wiki/Trie
[2] http://en.wikipedia.org/wiki/ISO_639-3

```
data
└── library
    ├── c
    │   └── e
    │       └── s
    │           ├── derived
    │           └── original
    ├── d
    │   └── e
    │       └── u
    │           ├── derived
    │           └── original
    └── e
        └── n
            └── g
                ├── derived
                └── original
```

Where we have only presented the Czech (ces), German, (deu) and English (eng) tries. In productive environments, PMSE can handle all of the 7000+ languages defined in iso639-3 this way.

If your environment has to cope with lots of languages, in order to be able to quickly visit the respective language on the command line, there is a shell alias `go` defined. This allows you to do

```
$ go ell
```

And you end up in the directory `/data/library/e/l/l/original/` (Greek original data).

In the original directories you can place your raw data files as you desire. because the derived directory has to maintain a quasi-mirror image of the original data structure, its stucture is slightly more complex, because one source file in *original* may correspond to several target files in *derived* (think e.g. about .zip archives in the original dir).

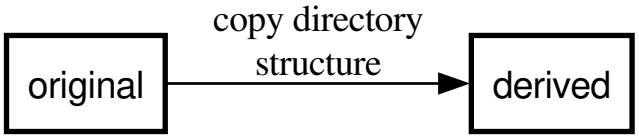

Figure 8.2: P_dmf directory structure

P_dmf will create similar structure in *derived* directory, but also will (eventually after several steps) convert the original document to a plain text encoded in UTF-8. The resulting text is placed in a directory called **lvl.last**.

Assume we have in the german original directory several yearly archives of PDF texts

```
$ ls /data/library/d/e/u/original/journal
```

```
2004.tbz2 2005.tbz2 2006.tbz2 2007.tbz2 2008.tbz2 2009.tbz2
2010.tbz2 2011.tbz2 2012.tbz2 2013.tbz2 2014.tbz2
```

where every of these archives contains several PDF files:

```
$ tar txf 2006.tbz2
```

```
2006/
2006/January.pdf
2006/February.pdf
2006/March.pdf
...
```

Then, after this script has processed these files, we will end up in the derived directory structure with something like this:

```
├── 2004.tbz2
│   ├── lvl.1
│   │   └── 2004.tbz2
│   │       └── 2004.tar.bz2
│   ├── lvl.2
│   │   └── 2004.tbz2
│   │       └── 2004.tar.bz2
│   │           └── 2004.tar.bz2
│   │               ├── 2004.tar
│   │               └── 2004.tar.gz
│   ├── lvl.3
│   │   └── 2004.tbz2
│   │       └── 2004.tar.bz2
│   │           └── 2004.tar.bz2
│   │               └── 2004.tar
│   │                   └── 2004
│   │                       ├── January.pdf
│   │                       ├── February.pdf
│   │                       ├── March.pdf
...
│   ├── lvl.4
│   │   └── 2004.tbz2
│   │       └── 2004.tar.bz2
│   │           └── 2004.tar.bz2
│   │               └── 2004.tar
│   │                   └── 2004
│   │                       ├── January.pdf
│   │                       │   └── January.txt
│   │                       ├── February.pdf
│   │                       │   └── February.txt
│   │                       ├── March.pdf
│   │                       │   └── March.txt
...
```

These `January.txt, February.txt, March.txt` files are what we were looking for.

No matter how many levels of conversion are required to end up with the desired target text[3], there will always be a `lvl.last` directory containing symbolic links to the final texts.

[3] if possible at all, as you may have placed e.g. photos in the original directory

Which means that if you are not interested in the intermediary data created during the conversion process, you can go to or point further processing tools to this directory.

8.3 Input Formats

P_dmf is able to handle several formats to convert input data into txt:

compressed files	7z ace ar arc arj bz2 cab gz lha lzma lzx pbz pbz2 pet rar rpm rz sea shar sit tar tar.bz2 tar.gz tar.rz xar xz zip zoo
doc files	doc docx fodt chm htm html htmlz odt ott rst rtf snb stw sxw troff txt uot wpd xml
pdf related	djvu dvi pdf ps tex texi
ebooks	azw epub fb2 lit lrf mobi pdb tcr txtz
other formats	cpio png

And possibly others. The real value add to `P_dmf` is its inferential capability to apply a chain of conversions to achieve a result even if your input data are nested archives and/or formats that need intermediary processing until a UTF-8 encoded text can be obtained.

8.4 Dependencies

P_dmf uses several available external converters. The script will warn you, if some of these auxiliary tools are missing on your system.

If you want to know the supported or available converters on your system, use the `--conv` option. If you want to know which are missing on your system, call

```
$ PM_CONVERTOR_WARNINGS=1 P_dmf -?
```

This will list supported, but not installed converters:

```
For support of arc2_ conversion install a tool used by PMLIB...
For support of arj2_ conversion install a tool used by PMLIB...
For support of chm2pdf For support of arc2_ conversion ...
For support of arj2_ conversion install a tool used by ...
For support of chm2pdf conversion install a tool used ...
```

Missing dependencies may be also tested (and installed) by `pm_install` script which is placed at `/opt/PetaMem/bin/active`. Just call

```
$ perl pm_install -testonly
```

and the installation process will be driven automatically.

8.5 Input Texts, Encoding

P_dmf checks files for encoding, but it is only a heuristic guess. When the encoding is not recognized, text is removed from a processing queue. Although P_dmf is able to change

encoding and to ”repair” the text[4], these functions are limited.

If the pre-defined conversion methods do not fit the requirements on the file processing, it is possible to specify a configuration file with a complete set of instructions - including a call of a external application. See section 8.6 below.

8.6 Wikimedia Processing

Wikimedia files are fetched by P_daf (see 7) in PMSE as `xml.bz2` dumps. The Wiki file must have a specific name containing a date in a format of `YYYYMMDD`. The absolute path of the file may be e.g.:

```
/data/library/e/n/g/wikimedia/wikipedia/eng-20110901.xml.bz2
```

P_dmf will load the newest file, unpack it, split it into single articles and create a trie structure in the *derived* directory according to names of the articles. This feature is quite different from the standard procedure (conversion from one format into another) and therefore it needs a special code path.

This code path is specified via `.dmf.info` configuration files. These files should be included in a directory (or a parent directory) with the source files - that means somewhere in the *original* directory.

8.6.1 Wikipedia Configuration File

The configuration file should be stored in a text format. See the example `.dmf.info` example below.

The whole config file is snippet of Perl code. The main frame of the config file is an array (reference - see enclosing square `[]` brackets). Each position in the array holds a hash (reference) `{}` - the hash contains information for processing of specific file. The hash may contain several key/value pairs:

order defines the sort order of sections if instructions for multiple files are defined. (The main array contains more than one hash reference.)

regex defines a regular expression which is matched against all source files in a given (sub)directory. If the name of the source file matches the regexp, the source file is recognized as a target of this configuration file and the operations specified in *tasks* key will be processed on the specified source files.

tasks defines a hash which itself may contain three keys:

process Replaces the generic conversion chain with specific instructions

prehook Allows to modify the processed file before the conversion.

posthook Allows to modify the processed file after the conversion.

The type of the conversion chain and the defined hooks are independent.

The *process* key in the previous example contains instructions for the conversion. It can be briefly described in a few steps: We check the date in the filenames. This is handy when multiple Wikimedia dumps are present in the same directory, because we want to process

[4] See section 15 for details.

Table 8.1: .dmf.info example

```
[
    {
        order   => 1,
        regex   => qr{/original/wikimedia/\w+/\w{3}\-\d{8}.xml},
        tasks   => {
            process => sub {
                my $args_hr = shift;
                my $w = $args_hr->{file};

                my $newest = 1;

                # if filename contains a date
                if (my($s, $version, $e) = ($w =~ m{(.*)((?:1|2)(?:9|0)\d\d(?:0[0-9]|10|11|12)(?:[0-2]\d|30|31))([^/]*)}xms)) {
                    my @all_file_versions = glob("$s*$e");
                    my @all_dates = sort { $b <=> $a } map {s{\A$s(.*)$e\z}{$1}; $_;} @all_file_versions; ## no critic
                    $newest = 0 if $version != shift @all_dates;; # return newest version
                }

                if ($newest) {  # no version
                    # parse the filename to get name of current file
                    $w =~ m{^(?<path>.*?)(?<file>\w{3}-\d{8}\.xml(?:\.bz2)?)$}xms;
                    $w = realpath($w);
                    # the absolute path to the file /data/library/i/n/a/original/wikimedia/wikipedia/ina-21130909.xml.bz2
                    my $file = $+{file}; # name of the file: ina-21130909.xml.bz2

                    my $out = $w;
                    $out    =~ s{original}{derived}xms;
                    $file   =~ s{\.bz2}{}xms; # strip suffix of the archive

                    my $cmd = "P_dm_wiki --action txt=$file --in $w --out $out/lvl.last/$file --fullnames";
                    pmse_report("Performing $cmd\n", 3);

                    `$cmd`;    # exec the command

                    pmse_report("The split of $file is finished\n", 3);
                }
            },
        },
    },
]
```

only the latest dump.[5] Then is prepared the resulting path in *derived* directory. Finally, the Wikipedia dump is unpacked and segmented by `P_dm_wiki` script.

8.6.2 Further Processing

P_gnp is able to read all files from given directory and process them at once. So, if you want to obtain a word list, or a list of n-grams from the whole Wikipedia, just specify the input directory - like e.g.:

```
$ P_gnp --in $ PMCORP_ROOT/e/n/g/derived/wikimedia/wikipedia/ <other opts>
```

8.7 Examples

Documents processing

Process all newest documents in *wikimedia* directory. The `--in` option is specified as a glob, thus the *wikimedia* directory is searched as a shallow structure.

```
$ P_dmf --in original/wikimedia/*
```

Process all files in *original* directory. Use P_ici for parallel processing and recursive descent.

```
$ P_ici --recurse --parallel --cpu 8  'P_dmf --in [%f]' original
```

[5] The unpacking and segmentation of Wikipedia requires hundred thousands of directories and gigabytes of disc space.

Chapter 9

P_dmp: Distance Measures Processor

`P_dmp` calculates distance measures for pairs of N-grams. You can use about cca. twenty distance metrics. See section 9.4 that describes these in detail.

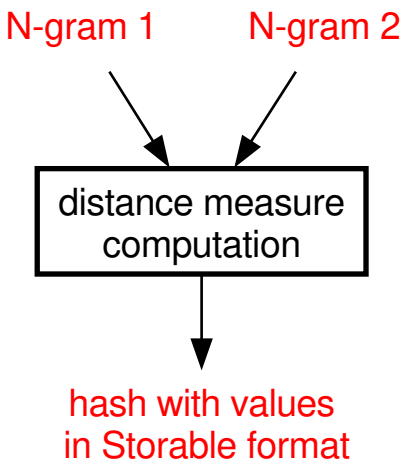

Figure 9.1: P_dmp schematic overview

```
$ P_dmp -?
```

9.1 Reference

PMSE Distance Measures Processor

USAGE P_dmp [options]

OPTIONS

–bulk <**filename**>

Read options `--ngrams` and `--distance` and distance parameters from given bulkfile.

–distance <**distance_name**=**params**|**?**>

Will count values of the specified distance measure for the given N-grams.

To list available measures:

```
P_dmp -distance ?
```

Available common parameter is `normalized`. If it is true, resulting distance is rescaled and has values in interval [0, 1].

There are few distances having specific optional parameters. These are:

mahalanobis: matrix

Matrix has to be regular matrix written in the form as:

```
[[1, 0], [0, 1]]                 for 2-dimensional space
[[1, 0, 0], [0, 1, 0], [0, 0, 1]] for 3-dimensional space
```

etc. If unit matrix is specified, mahalanobis comes into euclid.

minkowski: p

Domain of `p` is interval (0, Inf].

In case `p=2` distance comes into Euclid. For `p=1` we are receiving manhattan distance.

tversky: coef12 coef21

Both parameters are nonegative reals and at least one is positive.

For coef12=coef21=1 tversky becomes tanimoto. For coef12=coef21=0.5 tversky becomes dice.

jaro_winkler: prefix_lenght_min prefix_weight

0<`prefix_weight`<1/4, 0<`prefix_lenght_min`<=4

–ngrams <**ngram1**> <**ngram2**>

Contains N-grams for which a mutual distance will be computed.

–out <**filename**>

Defines the output directory. If `--out STDOUT`, the output will be printed to STDOUT.

–separator <**str**>

Character(s) separating tokens in N-gram. Empty string by default.

9.2 Examples

Example of use

```
$ P_dmp --distance 'levenshtein=normalized=1' --ngrams 'black dog barks' \
> 'white dog barks' --out STDOUT --separator ' '
```

This will calculate normalized Levenshtein distance between trigrams `black dog barks` and `white dog barks`

The output looks like:

```
 'levenshtein' => {'params' => {'normalized' => 1, 'l1' =>
['black','dog','barks'], 'l2' => ['white','dog','barks']},
 'value' => '0.333333333333333'}
```

Similarly, we could move options to bulk file. Let's have following file `bulk`:

```
[levenshtein]
l1=black dogs barks
l2=white dogs barks
normalized=1
```

We could call script like this:

```
$ P_dmp --bulk bulk --out STDOUT --separator ' '
```

9.3 Q&A

How to display the output?

The option `--out` defines the name of the outfile, which is stored in Storable format. (Use `--out STDOUT` to print the result on STDOUT.) To convert the output to human-readable text use the `P_dvf` or the Perl Data::Dumper [1] module.

9.4 Distance Functions Characteristics

The following distances are supported by `P_dmp`. The most of them are metrics (i.e. they are following axioms of identity, symmetry and triangle inequality) when the `normalized` option is used. Several distances are not metrics (`renkonen` distance breaks axiom of identity, `tversky` distance breaks axiom of symmetry, `jaro_winkler` breaks axiom of triangle inequality etc.).

For all the following paragraphs let's suppose we have N-grams (i. e. vectors) p and q. Let's denote p_i as i'th token of N-gram p, and q_i as i'th token of N-gram q.

Futhermore let $\#p$ denote the number of tokens in N-gram p (i. e. $\#p = n$) and let $\hat{\#}p$ denote the number of unique tokens in N-gram p (i. e. $\#p \geq \hat{\#}p$). Similarly for the others vectors/sets.

We will also need unions and intersections. Hence let's denote I as the set of tokens at the intersection of p and q and U as the set of tokens at the union of p and q.

Another terminology is need for the distances between frequency-ordered N-grams. In fact, a frequency-ordered N-gram is a histogram. Let $f(x)$ denote the position of the token x in the vector/histogram p. Similarly, let $g(x)$ denote the position of the token x in the vector/histogram q if it exists. Position of x needn't to exists in p. In this case let's $g(x)$ denote $\#q$.

[1] http://search.cpan.org/~smueller/Data-Dumper-2.131/Dumper.pm

Block distance

Let's denote P as the number of elements of p in i and, similarly, let Q denote the number of elements of q in i. Then we can define the Block distance as follows:

$$D_{block}(p,q) := \frac{\#p + \#q - P - Q}{\#p + \#q}.$$

BrayCurtis distance See Dice distance.

Canberra distance

$$D_{canberra}(p,q) = \sqrt{\sum_{i=1}^{n} \frac{|f(p_i) - g(p_i)|}{|f(p_i)| + |g(p_i)|}},$$

City block distance See Manhattan distance.

Chebyshev distance

Let p and q are frequency-ordered histograms of the same dimension. Then we define:

$$D_{chebyshev}(p,q) := \max_i(|f(p_i) - g(p_i)|).$$

Chess-board distance See Chebyshev distance.

Correlation distance

In fact this distance equals sample correlation coefficient. See PMLIB::Stochastic::Sample::sample_correlation.

Cosine similarity

Both vectors must be of the same length $\#p = \#q = n$.

$$D_{cosine}(p,q) := 1 - \frac{\sum_{i=1}^{n} p_i q_i}{\sqrt{\sum_{i=1}^{n} p_i^2}\sqrt{\sum_{i=1}^{n} q_i^2}}$$

Czekanowski distance

$$D_{czekanowski}(p,q) := \frac{2\#I}{\hat{\#}p + \hat{\#}q}$$

Damerau-Levenshtein distance

The Damerau-Levenshtein distance is calculated by counting the minimum number of operations needed to transform one string into the other. An operation is defined as an insertion, deletion, or substitution of a single character, or a transposition of two adjacent characters.

See also Levenshtein distance, which works in a similar process, except that it doesn't include transpositions.

Dice distance

$$D_{dice}(p,q) := \frac{2\#I}{\#p + \#q}\,.$$

It is special case of Tversky distance for $\alpha = \beta = \frac{1}{2}$.

Discrete Distance

$D_{discrete}(p,q)$ is 0 if p and q are not identical, 1 otherwise.

Edit distance See Levenshtein distance.

Euclidean distance

The Euclidean metric is probably the best known distance. The Euclidean distance between points p and q is the length of the line segment connecting them.

Here is the form for frequency-ordered histograms of the same dimension:

$$D_{euclidean}(p,q) = \sqrt{\sum_{i=1}^{n}(f(p_i) - g(p_i))^2}\,.$$

It is a special case of the Minkowski and Mahalanobis distance.

Hamming distance

the Hamming distance, between two ngrams of the same dimension, is the number of positions at which the corresponding symbols are different.

Jaccard index

The Jaccard index is used for comparing the similarity and diversity of sample sets. The index is given by the formula:

$$D_{jaccard}(p,q) = \frac{\#I}{\#U}\,.$$

Jaro distance

$$D_{jaro}(p,q) = \frac{1}{3}\left(\frac{m}{\#p} + \frac{m}{\#q} + \frac{m-t}{m}\right),$$

where

- m is count of tokens occurring on similar positions in both n-grams, i. e. $m = \#\left\{x : |f(x) - g(x)| < \lfloor\frac{\max(\#p,\#q)}{2}\rfloor - 1\right\}$. Let M denote set of these tokens.
- t is half the number of transpositions for tokens in M.

Jaro-Winkler distance

It is a variant of the Jaro distance metric with 2 extra parameters: we have k as a `prefix_length_min` parameter and l as a `prefix_weight` parameter.

$$D_{jaro\_winkler}(p,q) = D_{jaro}(p,q) + kl(1 - D_{jaro}(p,q))$$

Kulezinski distance

$$D_{kulezinski}(p,q) = 1 - \frac{\#I}{\#U}$$

Levenshtein distance

The Levenshtein distance, sometimes called the *edit distance*, is a string metric for calculating the difference between two sequences. It is defined as the minimum number of edits needed to transform one ngram into the other, with the allowable edit operations being insertion, deletion, or substitution of a single character.

Mahalanobis Distance

The Mahalanobis distance is based on correlations between variables by which different patterns can be identified and analyzed. It calculates the similarity of an unknown sample set against a known one.

If we let p and q be frequency-histograms of the same dimension. And we also let v denote a vector $(f(p_i) - g(p_i))_{i=1}^{n}$. Given a regular matrix S we can obtain the distance by using the following formula:

$$D_{mahalanobis}(p,q) = \sqrt{v^T S^{-1} v}$$

We can observe that Mahalanobis distance comes to Euclidean if S is unit matrix.

Manhattan Distance

$$D_{manhattan}(p,q) = \sqrt{\sum_{i=1}^{n} |f(p_i) - g(p_i)|}\,,$$

This is a special case of a Minkowski distance.

Minkowski Distance

The Minkowski distance is a form of geometry in which the usual distance function, or metric of Euclidean geometry, is replaced by a new metric in which the distance between two points is the sum of the absolute differences of their coordinates.

$$D_{minkowski}(p,q) = \sqrt{\sum_{i=1}^{n} (f(p_i) - g(p_i))^p}\,,$$

where p is given as a parameter.

Needleman-Wunsch similarity Returns number of one-element matches from optimal local alignment between two vectors according do Needleman-Wunsch algorithm.

Overlap distance

$$D_{overlap}(p,q) = \frac{\#I}{\min(\#p, \#q)}\,.$$

Renkonen distance

Let $r_v(x)$ be the relative occurence of the token x in the ngram v. If we suppose $\hat{\#}p = \hat{\#}q$, then the Renkonen distance is given by the following formula:

$$D_{renkonen}(p,q) = \sum_{x \in u} \min(r_p(x), r_q(x))\,.$$

Russel-Rao dissimilarity

Both boolean vectors must be of the same length.

Let's denote n_{ij} as a number of corresponding pairs of elements in p and q respectively equal to i and j.

$$D_{russel\_rao}(p,q) = \frac{n_{10} + n_{01} + n_{00}}{\#p}$$

Smith-Waterman similarity Returns number of one-element matches from optimal local alignment between two vectors according do Smith-Waterman algorithm.

Sokal-Sneath dissimilarity Both boolean vectors must be of the same length.

Let's denote n_{ij} as a number of corresponding pairs of elements in p and q respectively equal to i and j.

$$D_{sokal\_sneath}(p,q) = \frac{2(n_{10}+n_{01})}{n_{11}+2(n_{10}+n_{01})}$$

Sørensen distance See Dice distance.

Tanimoto distance

Tanimoto is a special case of a Tversky distance for $\alpha = \beta = 1$.

Tversky distance

Let $s(a,b)$ denote those tokens from the N-gram a which are not in the N-gram b. The Tversky distance is an asymmetric similarity measure which compares a variant to a prototype.

$$D_{tversky}(p,q) = \frac{\#I}{\#I + \alpha\#s(p,q) + \beta\#s(q,p)}$$

See both the Tanimoto and the Dice distances for important special cases.

Typo distance

Typo (typographical) distance is a combination of levenshtein and euclid distance measures. Given two strings, we use levenshtein distance to detect the place in the sequence where edit operation will happen, then we count euclidean distance of the characters participating on the operation.

Input parameter is the keyboard model and the key-map for given language. Various models are available. See PMLIB::Metric::Typo for more information.

Yule dissimilarity Both boolean vectors must be of the same length.

Let's denote n_{ij} as a number of corresponding pairs of elements in p and q respectively equal to i and j.

$$D_{sokal\_sneath}(p,q) = \frac{2(n_{10}+n_{01})}{n_{11}n_{00}+n_{10}n_{01}}$$

Chapter 10

P_dvf: Data Visualization Framework

The `P_dvf` script enables you to read and convert output of PMSE scripts into a human-readable format. This is necessary because the PM scripts outputs Perl native data structures which can be difficult to read easily. You can convert the data to various data-types. `P_dvf` also supports formatting, sorting and filtering of the output data.

You can load in data stored as Storable, text (even data printed with Dumper) and data serialized with YAML. Data generated in PMSE is stored as a PMSE object in Storable format. Each object has specific methods for visualization and post-processing depending on the nature of the data.

Data imported outside of PMSE will be handled with the most basic class: PMSE::Visualize. This class allows merely basic visualization methods and storage. But if you know your data correspond with one of the visualization class of PMSE::Visualize, you may specify type of object (see reference: 10.1).

```
$ P_dvf -?
```

10.1 Reference

PMSE Data Visualization Framework

USAGE P_dvf [options]

P_dvf is a script intended for displaying results of low-level PMSE data-mining processes.

OPTIONS Unlike to other PMSE scripts, options may differ depending on the type of input data. Some objects support some options and some don't. Each object is aware about methods, that may be used publicly, so when some P_dvf option is used in a wrong way, the script will throw out a warning. The important thing to know is that all PMSE objects have built-in documentation. When invoked, the self-documentation will provide info, how to handle given object in P_dvf.

Below is a complete list of options thay may be used with P_dvf.

–doc all|<**option**>

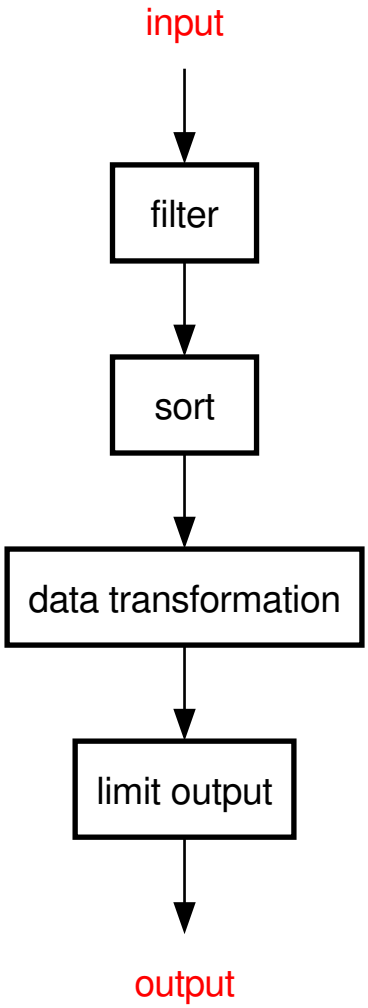

Figure 10.1: P_dvf schematic overview

Will list info about loaded object. If `all` parameter was chosen, script will provide complete overview about options that may be used with given object. You may also specify one single option.

–filter <**code**>

The filtering is done via a perl code:

```
'$key ~= m{\d+}xms'
```

Will remove all keys (tokens) matching given regex.

```
'$val == 60'
```

Will remove all occurrences of given value. Note: filtering of Contingency object will not cause the re-calculation of the values in the table.

–in <**filename**>

Defines an input file. Option `--itype` defines various available input data types.

–itype <**type**>

Format (type) of input data. Valid types are:

```
sbl     Perl data structure
        (default type)

dumper  Data::Dumper style printed data
text    plain text format
yaml    YAML-style data
```

–istruct <**object type**>

This option may be applied when you import external data structure into P_dvf. Let's say you have a histogram in textual format, like this:

```
AA    3.5
BB    0.22
XX    1
...
```

By default, this structure will be visualized (and dumped) by the PMSE::Visualize module, which provides basic methods only. When the input structure is a histogram, it may be specified with this option. Data will be treated then with appropriate module (e.g.: with PMSE::Visualize::Histogram).

–otype <**out-type**>

Type of output data. P_dvf adopted the object oriented approach to visualization, therefore exist a range of methods for given object. Some of them are generic and some are specific

Generic methods are

```
graphic     SVG
pdump       Data::Dumper output
storable    Storable (binary) file
text        plain text
yaml        YAML language
```

The SVG picture is created according to pre-defined options which are different for various objects.

Specific methods for object:

```
Binary tree (categorization output)
  newick
  spreadsheet
```

–out <**filename**>

Defines the filename for the output to be written to. If not defined, the output is printed to STDOUT.

–sort <**string**>

```
The sorting string may be:

+val  ascending
```

```
-val  descending
+key  alph. ascending
-key  alph. descending
```

10.2 Examples

Filter usage

Keys passed through positive filter:

```
_sort=$key !~ m{\d+}xmsg
sort_=$key !~ m{\w+}xmsg
```

1. Pre-sort filtering: data will be filtered before sorting. Unless the key contains a number, it is deleted. In other words: Only keys with a number will pass through.
2. Post-sort filtering: set on sorted data, only keys comprised from alphanumeric characters will pass through.

Negative filter on keys:

```
_sort=$key =~ m{\pP}xmsg
sort_=$key =~ m{\d+}xmsg
```

1. Pre-sort filtering: keys which contain a punctuation mark will be deleted.
2. Post-sort filtering: keys which contain a number will be deleted.

Filtering values:

```
_sort=$value > 15
sort_=$value < 5 and $value > 15
```

1. Pre-sort filtering: values greater then 15 will be deleted.
2. Post-sort filtering: values lower than 5 and greater then 15 will be deleted.

When we say that keys or values will be deleted we mean that the entire hash entry will be deleted.

Convert Text to Perl Data Structure

Consider the case where you have a printed data structure stored in a text file:

```
$VAR1 = {' ' => 1078,
         ' A' => 72,
         ' Ach' => 6,
         ' Aha' => 1,
         ' Aj' => 1,
         ' Ale' => 39,
         ' Ani' => 1,
         ' Atheneum' => 1,
        };
```

Now you want convert it back to an internal Perl data structure:

```
$ P_dvf --itype dumper --in printed\_text.txt --otype storable \
> --out storable_file
```

If you need the text in human readable form, just omit the `--otype` option, and the output will be automatically converted to text. For more information about converting to supported formats with see figure table below.

FORMAT	txt	pdump	YAML	Storable	dot	JSON	XML
txt	+	+	+	+	-	?	?
pdump	+	+	+	+	-	?	?
YAML	+	+	+	+	-	?	?
Storable	+	+	+	+	+	?	?
dot	-	-	-	+	-	?	?
XML	?	?	?	?	?	?	?
JSON	?	?	?	?	?	?	?

Chapter 11

P_gnp: Generic N-grams Processor

`P_gnp` is a powerful tool that covers a wide range of functions. It calculates and processes N-grams, creates contingency tables, as well as calculates MI-score and t-score. Also, it is capable to create ranks and compute key-words.

See the schema 11 for an overview of the process flow.

```
$ P_gnp -?
```

11.1 Reference

PMSE Generic N-grams Processor

USAGE P_gnp [options]

P_gnp is the core N-grams processing script in PMSE. It provides a wide range of N-grams evaluations.

The process workflow of P_gnp is as follows:

```
text(s) -> tokenization -> N-grams frequencies ->
contingency tables -> association measure score -> output
```

Throughout this processing chain, various hooks are deployed and support extensive data mangling (filters, transformations etc.).

OPTIONS

–bulk <**file**>

With the `--bulk` option you can define an INI file, which can be used for histogram / hash filtering and tokens processing. The INI file has a structure of '[section]' and 'name=value', the heredoc style can be used. 'Section' is the name of a procedure (process / ofilter), 'name' is name of a hook (the same as in `--process` and `--ofilter`). 'Value' stores the code, which will be executed on the result of an action.

–cluster [action1, action2 ... | ?]

Available actions are:

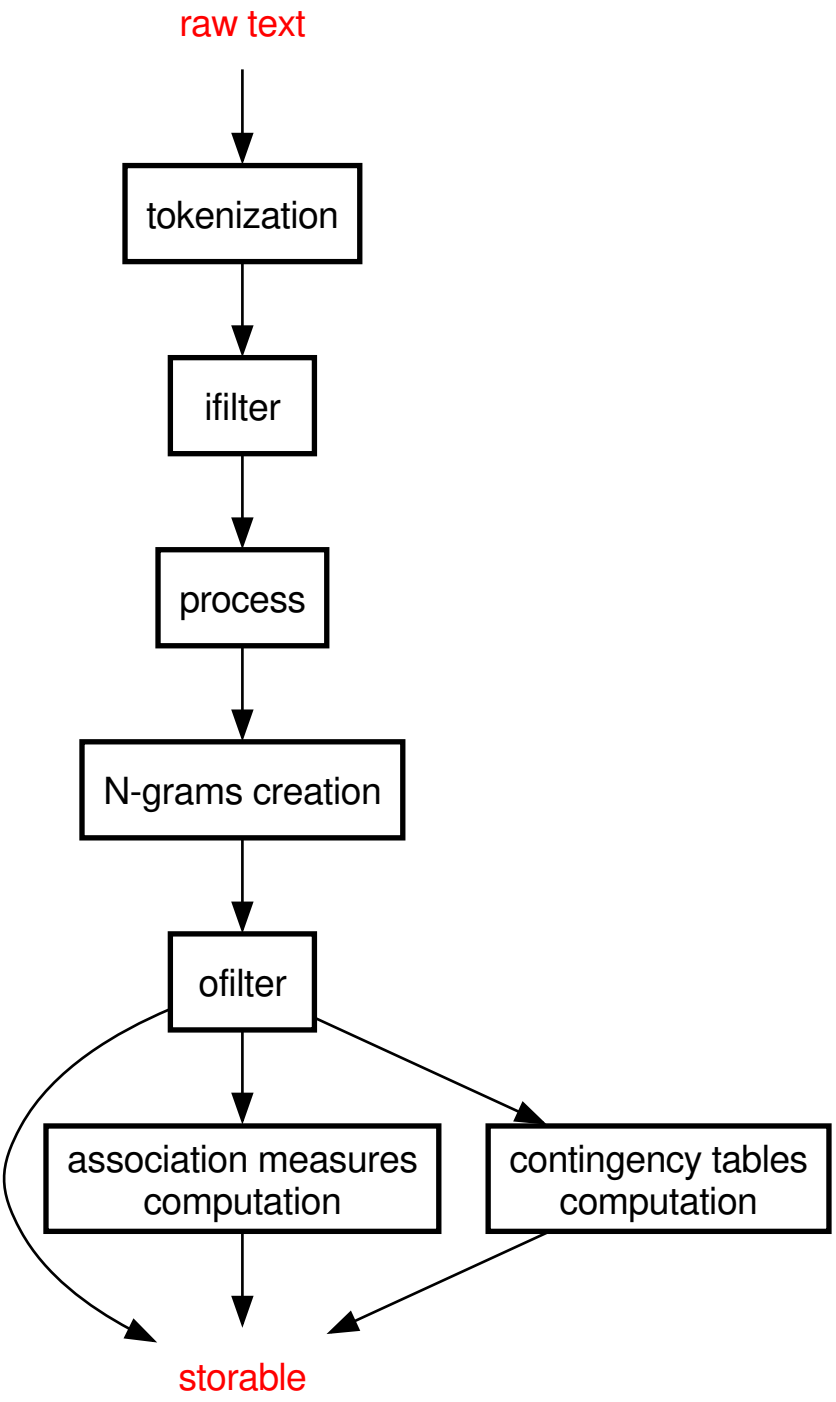

Figure 11.1: P_gnp - overview of the processes

```
count - results are n-grams counts
freq  - results are n-grams frequencies
prob  - results are n-grams probabilities
```

If no action is specified, the default action is output.

It is possible to specify several actions in one call.

–contingency

Computes contingency tables for N-grams.

Note: In case of a lot of N-grams it can take some time.

–delimiter <**regex**>

Delimiter has the form of a Perl regular expression. It enables the tokenizer to dissect the text into discrete tokens. If the user doesn't set his own value, the default delimiter from the PMLIB tokenizer is taken.

–histogram [<**action**>=<**filename**> | **?]**

We can count the distribution (histogram) for the result of each 'action'. The histogram is stored in file 'file'. Actions could be:

```
count      histogram of n-gram(s) occurrences
prob       histogram of n-gram(s) probability
freq       histogram of n-gram(s) frequency
measure    histogram of n-gram(s) measure(s)
```

Output file is stored in Storable format. When the hook is 'measures', a code referring to appropriate measure and dependence is prepended to the filename, because action 'measures' may contain multiple values.

–ifilter [<**type**>=<**regex**> | **?]***

This option may be provided multiple times (with different content for `type` of course) to define various filters, these are inserted at specific places during data stream processing. Valid values for ifilter <type> are:

```
+token    tokenizing step: matching tokens will pass
-token    tokenizing step: matching tokens will be blocked
+ngrams   n-gram processing step: matching tokens will pass
-ngrams   n-gram processing step: matching tokens will be blocked
```

<regex> may be any regular expression. Please take care to quote the whole expression, like '<filter>=<regex>', to prevent the shell modifying it.

–in <**filename**|**path**>

You may specify a file or a path to directory. If a filename is specified, only this concrete file will be processed.

If a path to directory was specified, the P_gnp will find recursively all plain text files with `*.txt` suffix and these will be tokenized in a parallel loop.

Note: in such case cross-text n-grams will be created unpredictably, because the order of input files will be random - that is a consequence of parallel processing.

–keywords <**filename**>

Outputs keyword weights. <filename> is hash in storable format representing frequencies/probabilities of reference corpus.

Interpretation: higher value for keyword means higher representativeness in corpus regarding to reference corpus.

–measure [<measure_name>=<args> | ?]

Computes requested measure for all N-grams.

The following measures are currently implemented:

```
mi    miscore
 t    tscore
```

Both MI-score and t-score accepts dependencies as a parameter. Dependencies are of the form

```
all
(nothing)
dep11|dep12|...|dep1N,dep21|dep22|...|dep2N,...,depM1|depM2|...|depMN
```

where 'depkl' is of the form 'number1 number2 ... numberp', where number is the i-th token of an N-gram.

If "all" is specified, all dependencies will be printed.

If no param is specified, all tokens are considered as independent. It is the same as if we would specify 1|2|3|...|n, i. e. 1|2 for bigrams.

Let's look at the 4grams example. If we specify: `--measure 'mi=1 2|3 4,1|2|3|4'`

P_gnp outputs for each N-gram:

```
1) MI-score for dependence between 1st and 2nd token
2) MI-score for no dependencies
```

Note: in dependencies, independent tokens could be omitted. This means it is sufficient to put '1 2' instead of '1 2|3|4'.

–ngram [<n-size> <window> <separator_character(s)>]

<n-size> must be a positive integer, `ngram` option defines the number of tokens from which an N-gram is comprised.

<window> is also a positive integer and defines the number of contextual tokens from which an N-gram is generated.

<separator> defines string to be used for separating tokens in an N-gram.

Default value is *2 2 ' '*. If you specify one of them, you have to specify all the others.

–ofilter [<hook>=<code> | ?]

Ofilter can be used on the result of each action, which is always a hash, thus with <code> we can affect keys and values. Hook denotes the specific part of the process where the `ofilter` is applied. Hook can be:

```
ngrams     filter hash with N-grams and their frequencies
_hist      filter result of action before histogram is created
histogram  filter hash with histogram values (note: keys and values
            are both numeric)
```

Code could have form of `$key =~ m{.*}xmsg` or `$value >=< number`. You can also insert the name of an INI file via the –bulk option.

–out <**directoryname** | **STDOUT**>

Defines the output directory. Results of all actions will be printed into correspondingly named files in this directory in the Storable format.

If STDOUT is specified instead of file, the Perl data structure will be printed directly on STDOUT in Data::Dumper format.

–process [<**hook**>=<**subst**> | **?]***

<subst> is a Perl substitution operator: ”s{pattern}{replacement}flags”, whereas ”pattern” is a regular expression, ”replacement” may be a string or even Perl code if the ”e” flag is given. For further info see e.g. http://perldoc.perl.org

<hook> may be one of the following:

```
_ngrams  first tokens processing before set is made
ngrams_  second tokens processing before set is made
```

A filter can be applied between both hooks. After the second processing of tokens is finished, the tokens-list is available, from which are generated N-grams.

–rank <**measure_name** | **?**>

Outputs rank of specified measure.

Measures have their own parameters (like dependencies in MI-score or t-score). If both **–rank** and **–measure** are specified, rank is computed for all satisfying measures. I. e. if **–measure ’miscore=1|2|3,1|2 3’ –rank=miscore** is specified, two ranks for MI-score are computed.

If **–rank** is specified and **–measure** is not, the default parameter for that measure (which is specified as parameter of **–rank**) is used.

11.2 Examples

Get basic N-grams list

```
$ P_gnp --cluster count --ngram 3 4 ' ' --out outdir --in corpus.txt
```

Will get 3-grams from the window of size 4 from ”corpus.txt” and store them as Perl data structures into the ”outdir” directory in the Perl Storable format. Tokens in each N-gram are delimited by whitespace.

Note: The number `N` must be a positive integer number, thus ’`--ngram -2.5`’ is not accepted.

Define more options for N-grams list

```
$ P_gnp --cluster count --ngram 2 3 ' ' --out outdir --delimiter '\s+' \
> --process '_ngrams=s{A}{B}xmsg' --bulk INI_file \
> --histogram 'ngrams=histogram' --ofilter 'histogram=$value < 5' --in corpus.txt
```

Computes bigram frequencies from a ’text’ window of size 3. The tokens in the text are split by white-space. The tokens in each N-gram are also delimited by whitespace. Before we get the hash with the N-grams, we change each token ’A’ to token ’B’. Where we originally had

'A D', we now have 'B D'. With `--bulk` we will load the INI_file with the next commands for processing. When the N-grams are finished, we create a histogram, which stores distributions of the N-gram frequencies. Because we don't want any values which occur less then 5 times, we use the `--ofilter` option to filter out these values.

Process multiple source texts

```
$ P_gnp --cluster count --ngram 2 4 ' ' --in /data/library/e/n/g/derived
```

Will find recursively all `*txt` files in `/data/library/e/n/g/derived` and process them at once.

Get contingency table for N-grams

```
$ P_gnp --contingency --ngram 2 2 ' ' --out outdir --in corpus.txt
```

Computes bigram contingency tables. For the options 'contingency' and 'measure', the separator must be given.

Get N-grams and MI-score

```
$ P_gnp --measure 'mi=all' --ngram 2 3 '<>' --out outdir --in corpus.txt
```

Computes MI-score for bigrams in a window of size 3, the tokens in each bigram are delimited like this: `word_A<>word_B`

```
$ P_gnp --measure 'mi=all' --ngram 3 4 '*' --out outdir --in corpus.txt
```

Computes MI-score for trigrams in window of size 4, resulting structure will contain all types of MI-score.

```
$ P_gnp --measure 'mi=1 2|3 4' --ngram 4 4 ' ' --out outdir --in corpus.txt
```

Computes MI-score for trigrams in window of size 4, resulting structure will contain the MI-score with the dependencies in 3rd and 4th tokens and in 1st and 2nd tokens.

11.3 Q&A

What is a 'window' in practice?

There is a sentence: `Fred is going to Wilma`, let's assume the user wants trigrams from window = 3. Then the first trigram will be: `Fred is going`, the second: `is going to` and the third: `going to Wilma`. If window = 4, we will have 4 tokens from which the trigram will be generated, in which case the total counts of trigrams will increase:

window = `Fred is going to`. Possible trigrams are: `Fred is going`, `is going to`, `Fred going to` and `Fred is to`.

What type of separator may be defined?

Everything you can represent with a regular expression.

```
'Fred is going'   (--separator = ' ' [whitespace])
'Fred*is*going'   (--separator = *)
'Fred<>is<>going' (--separator = <>)
```

N-grams in the output file are strange. They have different sizes.

Did you use the 'ifilter' option? It's possible that "trash" might be found within the tokens. For example: white spaces, punctuation, ends of lines etc. You could set

```
--ifilter '+token=\A\p{Alpha}+\p{Digit}*\z'
```

to get N-grams comprised from alpha-numeric characters only.

How do the dependencies work? What is MI-score and t-score for?

The MI-score for bigrams is computed as given by this expression:

$$\mathtt{MI} = \log_2 \frac{\mathtt{P(token1, token2)}}{\mathtt{P(token1)P(token2)}}$$

We can generalize this into a trigrams formula in several ways. If all the tokens are independent, we could use the following formula:

$$\mathtt{MI} = \log_2 \frac{\mathtt{P(token1, token2, token3)}}{\mathtt{P(token1)P(token2)P(token3)}}$$

In that case we don't use dependencies.

If we wanted to set a dependency between `token2` and `token3`, the formula would look like this:

$$\mathtt{MI} = \log_2 \frac{\mathtt{P(token1, token2, token3)}}{\mathtt{P(token1)P(token2, token3)}}$$

We add `--measure 'mi=2 3'`.

If we want to know about all kinds of dependencies, we have to add the option `--measure mi=all`.

Here is a formula for bigrams for the t-score:

$$\mathtt{t} = \frac{N\mathtt{P(token1, token2)} - \mathtt{P(token1)P(token2)}}{\sqrt{N\mathtt{P(token1, token2)}}}$$

Dependencies behave in a similar fashion in both t-score and MI-score.

However, the MI-score is not quite so accurate for low frequency N-grams. Some linguists recommend to filter the N-grams before computing the MI-score.

In the case of the t-score, high values are reached by synsemantic words (like *in*, *on*, *with*, *or* and so on) or by punctuation marks.

How to interpret the dependencies?

If no dependencies are set, we simply do a basic test for collocations.

In case we want to know more detailed information about current collocation, dependencies come in handy.

Let's say, we've found the trigram 'a b c' with high MI-score without dependencies. This means that some tokens of this 3gram are probably 'somehow' related. How related? That's where we should ask for dependencies.

If we switch the `--measure mi=all` option, we can see if the high MI-score in the independent case is caused by the bigram 'a b' (dependence '1 2'), or by the bigram 'b c' (dependence '2 3'), or even by 'a * c' (dependence '1 3'). Where all kinds of MI-score are high, this can signify that the entire trigram 'a b c' is a collocation.

If we have the 5gram 'a b c d e' we can also set more than 1 dependency. For example the dependencies 'a b' and 'b c d' could be specified altogether as '1 2' '3 4 5' (i. e. high MI-score in this 5gram could be caused by the bigram 'a b' and the trigram 'c d e' being collocations). We should note that with any outcome where 'n' is bigger than 3 or 4, we might well look for other kinds of MI-score.

The same techniques can be used for t-score.

Chapter 12

P_help: PMSE Helper

`P_help` will give you a basic reference about PMSE environment. Current version supports also listing of environmental variables.

```
$ P_help -?
```

12.1 Reference

PMSE Helper

USAGE P_help [options]

P_help provides generic help for PMSE environment.

OPTIONS

–pmroot <**path**>

You can specify PMSE root manually. Otherwise will be used a value of PMSE_ROOT variable.

–topic <**type**>|**?**

Choose topic of help. Available are:

```
list  will list all PMSE scripts, their function + version
env   will check and list environmental variables
```

12.2 Examples

Basic usage

```
$ P_help --topic list
```

Will list all PMSE scripts and will print brief information about their functions.

Chapter 13

P_ici: Intelligent Command Iterator

The purpose of this script is to provide a powerful, but easy to use, command iterator. While the shell provides several methods for using loops, it can be quite difficult to get simple repetitive commands done quickly as so called ”one-liners”. `P_ici` is meant to facilitate precisely this: Execute repetitive commands that change only a little in their input parameters.

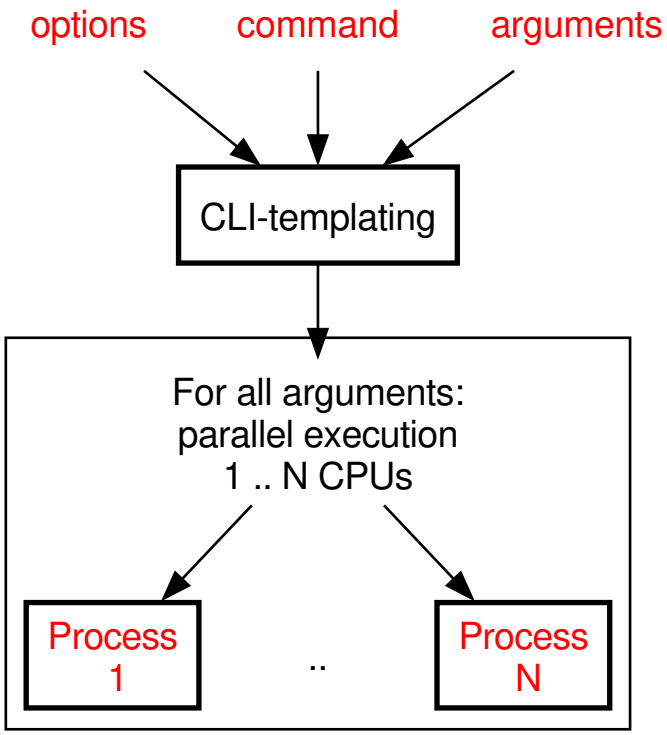

Figure 13.1: P_ici schematic overview

While the command may be used most often directly with files, it should also be considered a more generic tool. The arguments could be several tokens that have to be fed as input parameters to the input commands to be executed. Finally, if file searching, defining and filtering capabilities of the P_ici are not sufficient for your needs, you might consider combining this tool with the UNIX `find` utility.

```
$ P_ici 'echo [%f]' `find . <some exotic search params>`
```

There are some extensions to this idea, like parallelization, which go beyond the possibilities of shell iteration and allow you to make better use of your system resources (see a few examples in section 13.2).

P_ici allows to load two types of arguments: ”pure” arguments (where no semantic is assumed) and arguments related to path semantic (files or directories). The second type of arguments is loaded via `--in` option. The 'pure' arguments are interpreted in the frame of the command *as they are*.

On the other hand, files (specified with a glob or a recursive search) are transformed into their absolute paths before they are used as an argument for the command. Here is an example:

```
$ P_ici --in '*' 'echo [%f]'
```

```
/home/foo/file_a.txt
/home/foo/Documents
/home/foo/file_b.pdf
```

Will list all items (files, directories, links, etc.) that are matched by the '*' glob in the current working directory, with their absolute paths.

```
$ P_ici 'echo [%f]' *
```

```
file_a.txt
Documents
file_b.pdf
```

The shell will first find all visible files and directories in cwd, then it will send this list to `P_ici` as arguments. The result is a list of files with their paths relative to cwd.

It is very important to understand the difference between these two concepts and the need to escape shell expansion when you want `P_ici` to generate the list of files. Take for example the command

```
$ P_ici 'echo [%f]' --in *
```

which gives the output:

```
/home/foo/file_a.txt
Documents
file_b.pdf
```

Why is that? Because the shell expanded the unescaped glob '*' and send the arguments to `P_ici`, which will take these arguments and only the 1st one of them will be bound as parameter to `--in` and gets its file semantics, while the others will be passed as regular arguments as if you had done

```
$ P_ici 'echo [%f]' *
```

A Word About Parallelization

The available `--parallel` option allows you to spread the tasks to the available CPUs. In combination with the `--cpu` option you may even over- or undercommit the number of available CPUs (and thus the level of parallelization) overwriting the autodetection.

You may want to undercommit if you have IO-intensive tasks and your IO subsystem could not cope with a number of parallel tasks equivalent to the number of CPUs. On the other hand you may want to overcommit if you are developing and testing CPU-intensive parallelised tasks for a bigger system on a smaller system.

It is also noteworthy to say, that an automatic load balancing happens. If you have 500 tasks to process and 4 CPUs available, then each of the CPUs will get approximately 125 tasks if these are about the same runtime each.

If on the contrary all tasks have various runtimes, the CPUs will be given tasks to process as they become free. While `P_ici` itself cannot know the runtime of the task in advance, if you do, you can send longest-running tasks to be processed 1st and thus get an optimum overall runtime.

For many applications we can work under the assumption, that the longest files will also be those who take the longest time to process. Then, you can ensure by giving `--insort -size` that the biggest files are processed first.

```
$ P_ici -?
```

13.1 Reference

PMSE Intelligent Command Iterator

USAGE P_ici [options] cmd [argument(s)]

Where cmd defines the command that shall be applied to all arguments. cmd contains one or more unix/shell commands and one or more of these special placeholders:

```
[%\d] = the argument at index \d (given as positive integer)
[%#]  = the index of the current file (starting at 1)
[%b]  = basename of the current file (without last suffix)
[%d]  = directory portion of the current argument/file
[%f]  = the current file/argument quotemeta'd
[%F]  = the current file/argument raw
[%m]  = modified filename quotemetat'd
[%mb] = modified basename
[%md] = modified directory portion
[%ms] = modified last suffix
[%M]  = full modified filename unquoted
[%t2] = the current file/argument-trie as string
[%2t] = the current file/argument-string as trie
[%p]  = the current P_ici process-id
[%s]  = suffix of the current file/argument
[%t]  = current unix time in seconds since the epoch
```

```
[%u]  = current time in microseconds after [%t]
[%x]  = size of the current file/argument
```

This substitution does not happen if `--raw` option is given.

OPTIONS

–fatal

If this flag is set, execution of the issued command will stop on error. Else execution will continue (default).

–filter_name mod=<**repl**>|**neg**=<**regex**>|**pos**=<**regex**>

Define positive (everything that matches will pass), negative (everything that matches will not pass) and modification (everything will go through a replacement regex) filters for input files. Please be aware, that only files defined by `--in` parameter are actually considered for filtering. Arguments are unfiltered.

You can define all filters at once anywhere on CLI, where the 'mod' filter is applied after the positive and negative filters.

–filter_type [<**string**>**]**

Define a string with file test operators. By default this string is empty and all types of files are allowed as arguments. All Perl file test operators without the dash are allowed, e.g. 'f r s' means "only readable files with nonzero size". Please be aware, that only files defined by `--in` parameter are actually considered for filtering. Arguments are unfiltered.

–grace

If this flag is set and no arguments are given, the script will end gracefully.

–in <**filename**|**glob**>

Defines an input file or a glob, in which case all filenames that match will be considered for input. The glob must be escaped/quoted to prevent the shell from expanding it.

Arguments specified via `--in` option will be treaten as files / or directories, i.e. the path semantic will be expected.

If you want the input arguments to be 'pure', specify them on the end of the command as is suggested above.

–insort <**type**>|**?**

Defines the type of sorting for input files. Possible values for <type> are:

```
orig     sequence of input files as they come from shell (default)
shuffle  apply Fisher-Yates algorithm (unsort/shuffle file list)
+alpha   sort in ascending alphabetical order
-alpha   sort in descending alphabetical order
+size    sort in ascending file size (from smallest to biggest)
-size    sort in descending file size (from biggest to smallest)
+time    sort by file timestamp (from oldest to youngest)
-time    sort by file timestamp (from youngest to oldest)
```

–max <n>

Perform a maximum of `n` iterations.

–parallel

Execute commands in parallel, depending on number of cpus.

–raw

Do not interpret the command argument (ignore all `[%x]` sequences - if present - and treat them verbatim)

–recurse

Recursive descent when iterating arguments (files/directories/...). This option should be called together with `--in`.

–sleep <n>

Sleep `n` milliseconds between cmd calls in the main loop.

13.2 Examples

Find PDF Files

Find all PDF-Files (in the current directory and subdirectories) and convert them to ASCII-text (in parallel).[1]

```
$ P_ici --in '*' --filter_name pos='\.((?i)pdf)\z' --parallel --recurse 'pdftotext [%f]'
```

The PDF→TXT conversion will be performed in parallel, depending on the number of CPUs available - or given (`--cpu`).

In the above example, we are using a positive filter `--filter_name pos` to grab only pdf files from all files that were matched by '*'. We do this, because the filter we apply is case insensitive and thus we will get all files with endings like `.pdf`, `.PDF`, `.Pdf` and so on.

We could have achieved a similar effect more efficiently by directly globbing for PDF-files only like

```
$ P_ici --in '*{pdf,PDF}' --parallel --recurse 'pdftotext [%f]'
```

but this would omit weird cases like `.Pdf`, `.pDF`, `.pDf` etc. If you do know, that these do not occur in your data, the second form is more efficient. If you do not know the data or do not want to make any assumptions, the first form is preferred.

Identical Filenames in Subdirs

Next we will make the above example more specific. Let's consider we have files of identical names in subdirectories:

[1] `pdftotext` can be found as part of the poppler (see http://poppler.freedesktop.org/) package on most Linux distributions.

```
./web/dir1/listA.pdf
./web/dir1/listB.pdf
./web/dir2/listA.pdf
./web/dir2/bookxyz.pdf
```

If we use example 1, the second file called 'listA' will overwrite the first file of the same name when the text is extracted into the target directory, because we effectively flatten the directory structure. Thus we need to disambiguate 'dir1/listA' and 'dir2/listA' in the output. We can do that, for instance, by adding a timestamp, `[%u]`, to the output file name:

```
$ P_ici --in '*' --filter_name pos='\.((?i)pdf)\z' \
> --parallel --recurse 'pdftotext [%f] target_dir/[%u]-[%b].txt'
```

We recommend that you do not apply the template `[%#]` (an index of the current file being processed) when the option `--parallel` is in use. If you want to distinguish the identical filenames, this template will not help you, because each process evoked by `P_ici` obtains only one file from the source directory, thus the index is '1' for all files. Instead of `[%#]` you can use a timestamp `[%t][%u]`.

Advanced Filtering

You can apply all filters at once, where the available name filters (`--filter_name`) 'pos' and 'neg' will filter the given file names by applying the regular expressions given in a pass and block semantics respectively. The 'mod' filter is applied after these and will change the file names according to the replacement regular expression given.

Let's say you need to perform this complex filtering:

"Get all .pdf files which start with a lowercase character and have an uppercase character as the second character in their file name. From these filter out all those files that contain a 'x' (or 'X') anywhere in their filename. Finally, modify the resulting list to lowercase."

Your filtering arguments could look like this:

```
$ P_ici 'echo [%f]' --in '*.pdf' --filter_name pos='\A[[:lower:]][[:upper:]]' \
> --filter_name neg='[xX]' --filter_name mod='s{(.+)}{lc($1)}ge'
```

The reason we use the regex filtering instead of trying to use shell globbing, is because if our file names should be utf8, we would not catch all uppercase characters by a mere `[A-Z]` - thus omitting greek or cyrillic or other uppercase characters.

You can additionally filter by file type, where all Perl file test operators[2] are allowed. Please be aware, that the file test operators must be given without the preceding dash. Also, `P_ici` will ensure each operator is evaluated only once, even if given multiple times.

Modification Filtering

```
$ P_ici 'mkdir -p [%M]; mv [%f] [%M]' -filter_name \
> mod='s{.+\/(.).+\z}{uc($1)}e' -in 'alfons beta cecil'
```

[2] see http://perldoc.perl.org/functions/-X.html

Chapter 14

P_rer: Regular Expression Replacer

The `P_rer` script provides support for performing arbitrary and highly complex replacements on the contents of a set of files.

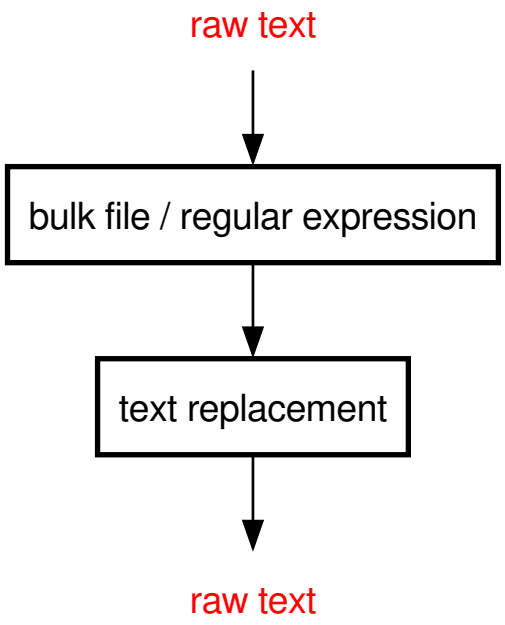

Figure 14.1: P_rer schematic overview

```
$ P_rer -?
```

14.1 Reference

PMSE Regular Expression Replacer

USAGE P_rer [options] regexp(s) file

At least one regular expression in the `form s{}{}` must be given. On the end of the command, exactly one file argument must be given, but you can enter a glob matching several files. Input and output may be specified also with otpions. You have to quote anything that contains white-space characters. Use 'P_ici' for more complex file specifications (albeit this imposes a performance penalty).

OPTIONS

–bulk <**file**>

If regular expressions are too cumbersome to enter with bash escapes and/or it is expected that you will need to recycle the replacement commands, you can specify an INI-style bulk file that will be used to define the replacements.

You can name the sections in any way you like, but the expected keys per section are <i> (mandatory and must not be empty), <o> (optional - default: empty string) and <m> (optional - default: 'xmsg').

You can use heredoc-style for the value definitions in all keys and you may define these keys multiple times per section, in which case they are concatenated. All other key names are ignored.

The resulting replacement is `s{i}{o}m`.

```
[section name]
  i = \N{INFORMATION SEPARATOR ONE}(.+)
  o = foo$1
  m = ms
```

will perform a `s{\N{INFORMATION SEPARATOR ONE}(.+)}{foo$1}ms` replacement on the text in question.

```
[multiline]
  i = a
  i = b
  o = x
  o = y
```

will perform a `s{ab}{xy}xmsg` replacement.

If several sections are given, the replacement order is defined by the alphabetical order of the sections.

–in <**file**>

Input file. When you use `--in`, you have specify also `--out` and vice versa. Prepending of file specification to the end of the command is not allowed.

You may use

```
P_rer <in> <out> <regexp>
```

or

```
P_rer <regexp> <file>
```

Combination is not possible.

–out <**file**>

Outfile. When you use `-out`, you should use also `-in` option. See `-in` for details.

–sect <**match_rx**>

You may optionally give a regular expression to define which sections are to be processed. By default, this are all sections that do not start with a '!'. (default is `qr{(?<!\A!)}`)

14.2 Examples

Replace occurrences

Replace all occurrences of 'computer' with 'notebook' in all text files in the current directory.

```
$ P_rer s{computer}{notebook}g '*.txt'
```

Remove trailing space

Remove all trailing space in the file `spatial.text`

```
$ P_rer 's{\s+\z}{}g' spatial.text
```

As the regular expressions make use of the Perl Regular Expression engine, you can also make full use of embedded Perl code within your regular expressions:

Edit timestamps

Replace all occurrences of `%d%` in all `*.time-stamp` files with the current date.[1]

```
$ P_rer 's{%d%}{scalar localtime()}ge' '*.timestamp'
```

[1] quoting the regular expression may be necessary if it contains spaces

Chapter 15

P_trt: Text Repair Tool

Similar to P_rer, `P_trt` is a tool for text manipulation. Unlike to P_rer, P_trt is working on more abstract level. Basic function of P_trt is a deformating of text. The most general function is `--action deformat`.

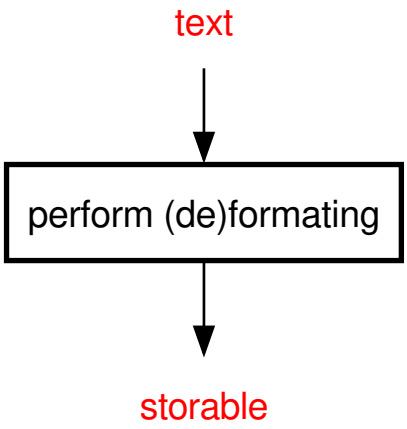

Figure 15.1: P_trt schematic overview

```
$ P_trt -?
```

15.1 Reference

PMSE Text Repair Tool

USAGE P_trt [options]

P_trt manipulates with text on more abstract level.

OPTIONS

–action <name>|?

Available actions are:

```
dehyphen
compress_whitespaces
deformat
remove_headline
repair_interpunction
trim_whitespaces
```

–in <**filename**>

Defines the input file name. If `--in STDIN`, the input will be read from STDIN.

To end the the input use `*END*` sequence.

–out <**dir**>

Defines the output directory. If `--out STDOUT`, the output Will be printed to STDOUT.

15.2 Examples

Repair short text

Consider two lines of text followed by 2 newlines:

```
Perl  is          cool.
( I think .       )
```

Now replace multiple inline white space characters to one single space and trim leading and trailing white space of a string:

```
$ P_trt --out STDOUT --in text.txt --action compress_whitespaces
```

```
Perl is cool. ( I think . )
```

Action `deformat` has in this case the same effect, because it replaces all newlines with spaces. Then the function performs compression of white space characters. (The end white space is trimmed, because nothig follows.)

Action `repair_interpunction` will try to make the text more coherent. It will group together the text, punctuation and brackets. Only one line will be affected in this example.

```
$ P_trt --out STDOUT --in text.txt --action repair_interpunction
> \end{verbatim}
>
> \begin{verbatim}
> (I think.)
> \end{verbatim}
>
> Here is a more complex example of interpunction repair:
>
> \begin{verbatim}
> Bla bla , bla.( Ha ,ha. ) # false
> Bla bla, bla. (Ha, ha.)   # correct
> \end{verbatim}
```

```
>
> Action \verb;trim_whitespaces; will trim white space characters on the beginning and end
> of the string.\\
>
> \begin{Verbatim}[frame=single]
> P_trt --out STDOUT --in text.txt --action trim_whitespaces
```

```
Perl  is           cool.
( I think .        )
```

More complex reparation of text may be performed by multiple use of P_trt.

Chapter 16

P_vls: Variable Length Splitter

`P_vls` is one of the helpers PMSE script. It is intended to cut specified amount of word types from a text file. P_vls converts the text file to a frequency list, sorts the list in descending order and then splits the list according to specified options.

Tokens delimiter can be specified.

It is possible to specify both absolute or relative number of the carved word types. Also, it is possible to cut a range. The output file is stored in Perl Storable format.

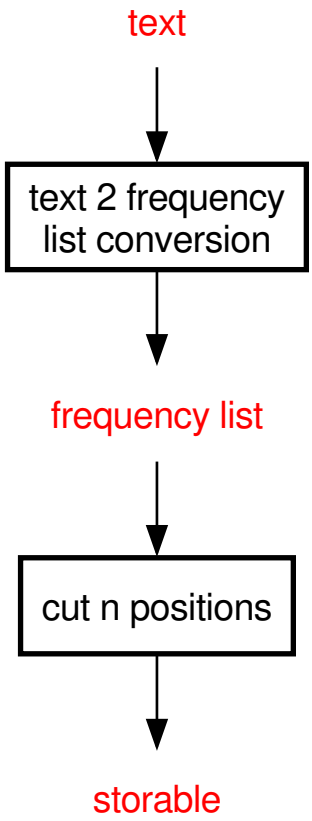

Figure 16.1: P_vls schematic overview

```
$ P_vls -?
```

16.1 Reference

PMSE Variable Length Splitter

USAGE P_vls [options]

P_vls extracts words from given files and gives a cut its order.

OPTIONS

–cut <**str**>

`str` is composed of bounds delimited by whitespaces. Each bound specify where to split the hash.

We returns a list of cuts from one bound to forthcoming one.

These boundaries could be specified as

```
absolute number (e. g. 40)
percent value (e. g. 40%)
increment (e. g. +10%, +50)
```

For example string '0 0 1 2 +1% 100%' for text of 1000 token types will be interpreted as following:

```
1st cut: first word
2nd cut: second word
3rd cut: third word
4th cut: from fourth to fourteenth word
5th cut: from fifteenth to the end (1000th)
```

–delimiter <**regex**>

String that will be converted to regex which delimits words.

–in <**filename**>

Input files.

If you specify glob, be sure to use apostrophes because of bash expansion.

–out <**directoryname**> **or** <**STDOUT**>

Defines the output directory. Results of all actions will be printed into correspondingly named files in this directory in the Storable format.

If STDOUT is specified instead of file, the Perl data structure will be printed directly on STDOUT in Data::Dumper format.

16.2 Examples

Basic information retrieval

```
$ P_vls --out STDOUT --in frequency_list.txt --cut '1000'
```

Will cut first 1000 words from frequency_list.txt and print them on STDOUT.

```
$ P_vls --out STDOUT --in frequency_list.txt --cut '10' '20'
```

Will cut word types from 10th to 20th position of the frequency list and print them on STDOUT.

```
$ P_vls --out STDOUT --in '*' --cut "+1% 0%" --delimiter '\s+'
```

Will cut e.g. first 10 tokens from a wordlist of 1000 tokens and will print the modified list in reverse order on STDOUT. The input is specified as a glob. We delimit tokens just by a white-space.

Chapter 17

PMSE: Tutorial

17.1 Learning by Example

The more functions PMSE offers for text processing the more difficult it becomes to learn how to use PMSE effectively. To get acquainted with the PMSE suite quickly, we have prepared tutorials with examples based on experience, of step-by-step PMSE script usage. You can find these on the following pages.

17.2 Corpora

Here are a few corpora we are using in examples.

17.2.1 C1

```
A natural nuclear fission reactor is a uranium deposit where
analysis of isotope ratios has shown that self-sustaining nuclear chain
reactions have occurred.
```

17.2.2 C2

```
Klasterec above Ohre is a nice city.
```

17.2.3 C3

```
192.168.0.1, 192.168.0.2, 192.168.0.1 192.168.0.3, 192.168.0.1, 192.168.0.3
```

17.3 P_csp Interactive

17.3.1 Basic Usage of –iact

Now we will show the basic usage of the `P_csp` command, step by step. This tutorial uses 'interactive mode' and should help you to get started quickly. There is also the 'CLI mode'[1], if you prefer. Interactive mode is easier to use than the CLI mode, but it is less powerful.

[1] CLI = command line interface

Complete functionality is achieved with CLI mode, although it is more complicated to learn. So while this is good place to start, by the time you have completed these exercises, you are encouraged to explore the power of the CLI.

We will use the *C2* corpus in this tutorial:

```
Klasterec above Ohre is a nice city.
```

```
$ P_csp --iact
```

First we have to insert the path to our corpus.

```
Insert path to the corpus :
pmse> c2
Wrong format (file is unreadable, empty or nonexisting)
```

This error message is generated because we made a mistake - the system is case sensitive. We have to type the path again, using the correct case:

```
pmse> C2

Insert directory or 'STDOUT' (default value: >P_csp.1569<):
pmse>
```

Now you can specify the path where the output directory is located. If you want to just display the output on the screen, type `STDOUT`. Note that entering no input (just pressing ENTER) will cause `P_csp` to create a local directory with the ID of the current run. This 'default' directory will be placed in the CWD[2].

```
Insert regex delimiting tokens (or ENTER to skip):
pmse>
```

This option is called the `delimiter`. It affects how the text is split. There is a default tokenizer in PMLIB, but you can define your own, let's take a look at an example (corpus *C2*):

```
Insert regex delimiting tokens (or ENTER to skip):
pmse> \s+
```

result:

```
120424-09:42:18.098449/1 - Output directory: STDOUT
'utcount' => {'a'     => 1,'above' => 1,'Klasterec' => 1,
              'city.' => 1,'is'    => 1,'Ohre'      => 1,'nice' => 1},
```

We can see the counts of tokens in the sentence. We used `\s+` - one or more spaces - to separate tokens in the sentence. The output looks almost OK, except for the sequence 'city.'. To clean up this token, and others like it, we must add a regex to define punctuation as the delimiter:

```
Insert regex delimiting tokens (or ENTER to skip):
pmse> \s+|\p{P}+
```

result:

```
120424-09:49:18.551789/1 - Output directory: STDOUT
'utcount' => {'city'      => 1,'a'  => 1,'above' => 1,
```

[2] Current working directory.

```
              'Klasterec' => 1,'is' => 1,'Ohre'  => 1,'nice' => 1},
```

Choice of the delimiter determines the concept of your tokens. Because of our choice of delimiter, you don't have punctuation marks and spaces among your tokens now. Also consider 'words' like "Mike's". With the current definition of delimiter you would get 'Mike' and 's' as separate tokens. If you want to have spaces and punctuation in your token list, define the delimiter like this:

```
Insert regex delimiting tokens (or ENTER to skip):
pmse> \b
```

Here we use a word boundary as the delimiting element. Now we will get:

```
120424-09:51:02.061125/1 - Output directory: STDOUT
'utcount' => {'city'      => 1,'a'  => 1,' '    => 6,'above' => 1,'.' => 1,
              'Klasterec' => 1,'is' => 1,'Ohre' => 1,'nice'  => 1},
```

The default PMLIB tokenizer is a little bit complicated to be described, but when we look at the output, it is the same as in the previous example. This is caused by the length of the text. The output would differ if we had longer texts with various characters.

You can also specify your tokens concept with other options, for instance: `ifilter` and `process`. `Process` is not implemented in the interactive mode, but you can specify it in CLI mode.

The list of tokens can differ in relation to the desired output, in other words in relation to the action you want `P_csp` to do:

```
Insert utcount, utprob or utfreq (you can specify more values
separated by whitespace):

pmse> utcount utprob
```

For information about each action take a look at the section 6.1. If you specify `utcount` and `utprob`, the output will be stored in the directory specified via the `out` option. The 'utcount; and 'utprob' files will be placed in this output directory - both in Perl Storable format. You can read them by running the `P_dvf` command.

```
Insert filters (the format of the pairs is key=value, pairs
are separated by ENTER):

pmse>
```

This option is called `ifilter`, because it filters tokens going into the process of statistical computation. If you found punctuation and spaces in your tokens list and you want to remove them, you should 'catch' them beforehand with a regexp in the `ifilter`:

```
pmse> -token=(\p{P}+|\s+)
```

The `key` part of the `ifilter` denotes the place where the filter is applied. Take a look at the figure 6.2. This figure represents the whole process of statistical computation. You might have one set of tokens, but you can run different operations within the set for each action.

You can also specify multiple filters. In interactive mode you are asked twice or more times for the values. If you just want one single filter, type ENTER and do not enter any further values.

There is also an 'ofilter' option available for `P_csp` in CLI mode only. The 'ofilter' acts on the output, which in our case is a hash with pairs `token => value`. If you do not want to filter the output of `P_csp` directly, you can also filter it using `P_dvf`. This can be useful when you need to display modified output, but do not wish to change the P_csp's output at all.

After you complete the `ifilter` options, an overview of your task is displayed:

```
we have following values:
  in => 'English',
  out => 'STDOUT',
  delim => qr/(?^:\b)/,
  action => ['utcount','utprob'],
  ifilter => {'-token' => qr/(?^u:(\p{P}+|\s+))/},
Is it correct? (default value: >yes<):
pmse>
```

Interactive mode then asks if the given values are correct. If you are not satisfied, type 'no' and you will be asked which values you want to change. Then you can re-enter each option again. If all options are OK, just type ENTER and your task will be computed.

17.4 P_gnp Interactive

This tutorial for `P_gnp` is designed to follow on from the tutorial for P_csp's interactive mode. Although these two scripts share some common elements of the architecture, `P_gnp` provides more functionality, and supports a wider range of input options.

Basic options, such as `in`, `out`, `ifilter`, `process` and `delimiter` are common for both scripts.[3] `P_gnp` also has an option called `cluster` which is identical to P_csp's `action` - in `P_gnp` determines `cluster` what should be counted as N-grams (their basic count, frequency and probability).

Taking a look at the `P_csp` interactive tutorial in section 17.3 is a good way to learn how to work with the basic options, and it is assumed the `P_csp` tutorial has been succesfully understood, before proceeding with this tutorial.

Options specific for `P_gnp` can be divided into two groups:

- options related to N-grams creation
    1. **size** – size of the N-gram
    2. **window** – size of context from which are ripped tokens creating the N-gram
    3. **separator** – element delimiting tokens in the N-gram
- options related to computation of (lexical) association measurements

[3] The delimiter option **always** defines how to split tokens in the text. Tokenization is the first process in P_gnp, and is followed by the creation of N-grams. For a graphical explanation of the flow of available actions, which might be easier to visualize, see figure 11.

1. **measure** – name of association measure
2. **rank** – N-grams get ranked according to specified association measure
3. **contingency** – compute contingency tables with probabilities of tokens and N-grams

To get detailed information about each of these options, see the chapter 11. Now let's take a closer look at the interactive mode. We will use the *C2* corpus again. We are looking to create a list of bigrams and we also want their count, MI-scores, and a list with MI-score ranked bigrams.

```
Insert the correct path to the corpus:
pmse> C2
VALUE: >C2<
```

You have specified the correct, case-sensitive, path to your file, right?

```
Insert directory or 'STDOUT' (default value: >P_gnp.2243<):
pmse> english_ngrams
VALUE: >english_ngrams<
```

Output will be stored in the 'english_ngrams' directory.

```
Insert regex delimiting tokens (or ENTER to skip):
pmse> \b
VALUE: >\b<
```

You want to delimit tokens in your text by word boundary.

```
Insert ngram size (default value: >2<):
pmse>
VALUE: >2<
```

If you want just bigrams, type ENTER - bigram is the default value.

```
Insert window size (default value: >2<):
pmse>
VALUE: >2<
```

Let's specify the contextual tokens that creates the N-gram.

```
Insert str separating tokens in ngram (default value: > <):
pmse>
VALUE: > <
```

The default value is a single whitespace. If you want something more specific, e.g.: `YX*XY`, type `*` and press ENTER.

```
Do you want to output contingency tables? (default value: >no<):
pmse>
VALUE: >no<
```

Counting of contingency tables is not particularly fast. Especially when you have a large corpus, which is why the default is 'no' here. (Just press ENTER.)

```
Insert count, prob or freq (you can specify more values separated by space):
pmse> count
VALUE: >['count']<
```

You chose a simple count of the N-grams. The other options are `prob` (probability) and `freq` (frequency).

```
Insert measurements and their parameters
(format of pairs is key=value, pairs are separated by ENTER):
pmse> miscore
pmse>
VALUE: >{'miscore' => undef}<
```

Parameter is mandatory, when you want to count the MI-score (or other association measure) for N-grams bigger than 2. If you want trigrams, you have to specify dependencies among the tokens creating the trigram. So - if you do not need to specify the dependencies, just type `miscore`.

```
Insert measure names for ranks (miscore or tscore)
(you can specify more values separated by space):
pmse> miscore
VALUE: >['miscore']<
```

If you want the list of bigrams ranked by MI-score, just type `miscore`.

```
Insert ifilter(s)
(format of pairs is key=value, pairs are separated by ENTER):
pmse> +token=\w+
pmse>
VALUE: >{'+token' => '\\w+'}<
```

With this token specification, the program will be searching for tokens comprised only from alpha-numeric characters. You can give multiple filters here. When you are satisfied with your choice, just type ENTER and the next option will be launched.

```
Insert substitution hooks (possible keys are _ngrams and ngrams_)
(format of pairs is key=value, pairs are separated by ENTER):
pmse>
VALUE: >{}<
```

This option in CLI mode is called `process`. You can modify your tokens here; for instance, you can transform all tokens to lowercase. Your current corpus contains just one sentence, so you probably do not need to change the tokens. However, if you wished to change all bigrams (tokens in bigrams) to lowercase, you could do something like:

```
_ngrams={\A(.+)\s}{lc($1)}xmse
```

```
we have following values:
  in => 'english',
  out => 'english_ngrams',
  delim => qr/(?^:\b)/,
  size => '2',
  window => '2',
  separator => ' ',
  contingency => '0',
  cluster => ['count'],
  measure => {'miscore' => undef},
  rank => ['miscore'],
  ifilter => {'+token' => qr/(?^:\w+)/},
```

```
  process => {},
Is it correct? (default value: >yes<):
pmse>
VALUE: >yes<
```

An overview of your task is now displayed. Is everything OK? If not, just type the option that need to be changed and adjust it accordingly. If everything went well, you should see a directory called `english_ngrams` with 4 files: `count`, `miscore_rank_1|2` `miscore_1|2` and `pmse-env`. The `pmse-env` file is a log, where you can see a record of your task. 'Count' is list of N-grams and their counts, `miscore_rank_1|2` are ranked N-grams and `miscore_1|2` are unranked N-grams with their associated MI-score values.

If you do not find it particularly user friendly to browse each file separately, you can use `$PMSE_BIN/samples/create_table.pl` to make a table:

```
$ perl $PMSE_BIN/samples/create_table.pl --in 'english_ngrams' --out STDOUT
```

This stanza will produce a CSV file:

```
data,count,miscore_1|2,miscore_rank_1|2
Klasterec above,1,2.58496250072116,1
nice city,1,2.58496250072116,3
is a,1,2.58496250072116,2
Ohre is,1,2.58496250072116,4
a nice,1,2.58496250072116,5
above Ohre,1,2.58496250072116,6
```

Which you can load into R or your favourite spreadsheet editor.

17.5 Categorization of EMA Texts

EMA is an abbreviation for *European Medicines Agency* [4]. EMA organization provides identical documents in parallel languages. All the documents are distributed in PDF format. In this example, we will describe how to

- download the documents
- convert them
- perform text categorization for each language

17.5.1 Fetch the Docs

Getting the docs is easy, because the P_daf script has already predefined hook called `ema`:

```
P_daf --fetch ema
```

P_daf will store the documents in `\$PMCORP\_ROOT/<iso>/original/ema` where `<iso>` is the iso-639-3 code transformed into a trie structure (see chapter 8). The default target is `$PMCORP\_ROOT` which you may change in the INI file (it's a line beginning with `store =`). E.g.: in the current EMA INI file[5] you will find line like this:

[4] http://www.ema.europa.eu/ema/
[5] Which is placed in `$PMSE_ROOT/cfg/daf.d/`

```
store = "$ENV{PMCORP_ROOT}/" .
        x2f_any2path({input => _1or3_639( $lang )}) .
        '/original/' . $pdf
```

To change the root, substitute `$ENV{PMCORP_ROOT};` with the desired path - e.g.: `/home/john/documents/ema`. The resulting path for i.e. English documents then will be: `/home/john/documents/ema/e/n/g/original/ema/`. (We will use this separate directory in order to keep this example easy.) For further info about how `P_daf` works read please the chapter 7. Let's presume we have fetched all the multilingual EMA documents successfully into a root directory `/home/john/documents/ema/`.

To see how many languages do we have, cd to that mentioned directory and type:

```
tree -L 2 -i -f * | grep './/.'
```

You should get a list of directories, which contain the ema docs:

```
b/u/l
c/e/s
d/a/n
d/e/u
e/l/l
...
```

Good. Now - all the EMA docs are stored in PDF format. We will call P_dmf and convert them into plain text files. We just need to find the 'original' directory for each language. While the P_dmf needs to get an absolute path of the input argument, we will do:

```
find `pwd` -name 'original' -exec P_dmf --base \
/home/john/documents/ema --in '{}' \;
```

This will create corresponding *derived* directories. Now it is easy to create simply shell script to execute the categorization:

```
#!/bin/bash

dirs=(f/i/n f/r/a n/l/d l/i/t
      h/u/n s/l/k s/l/v s/w/e
      s/p/a d/a/n d/e/u m/l/t
      i/t/a e/n/g e/l/l e/s/t
      p/o/l p/o/r )

for i in "${dirs[@]}"

do
    $PMSE_BIN/samples/categorization/categorization.pl\
    --root /home/john/documents/ema/$i --report 3 --cpus 8 \
    --vector frequent='count=200,distance=tanimoto,weight=1,\
    preprocess=1'
done
```

After execution of the script in all `/home/john/documents/ema/*/*/*/` directories should appear a folder called `runs` which holds the results of the categorization run.

Chapter 18

PMSE: Cookbook

18.0.1 Recipes for PMSE

Some recipes for several practical tasks take place in this section. We use the functionality of PMSE.

18.1 PMSE Crash Course

This section consists of a list of commands related to PMSE environment. Purpose of the list is to provide a quick reference on the most relevant topics / functions. You will find a detailed explanation of the commands further in the documentation.

File modifications

```
P_rer 's{replace}{with}g' file

P_trt --action 'repair_interpunction' --in filename  --out STDOUT

P_vls --in textfie --out STDOUT --cut '1 +10% 10% 90%'
```

Toolchain: decompression of an archive, extraction of n-grams

```
P_dmf --in $PMCORP_ROOT/e/n/g/original/archive.tgz

P_gnp --in $PMCORP_ROOT/e/n/g/derived/archive.tgz/lvl.last/\
archive.tgz/archive.tar.gz/archive.tar.gz/archive.tar/a.txt\
--ngram 3 3 ' ' --measure 'mi=all' --measure\
'tscore=1|2|3,1 2|3' --rank miscore --cluster count --cluster prob\
 --keywords reference_corpus.sbl

$PMSE_BIN/bin/samples/create_table.pl --in P_gnp.out\
 --out table.csv
```

Extracting a wordlist

```
P_csp --in $PMCORP_ROOT/e/n/g/derived/archive.tgz/lvl.last/\
archive.tgz/archive.tar.gz/archive.tar.gz/archive.tar/a.txt\
--action utprob --out STDOUT
```

Categorization toolchain

```
P_dmf --in $PMCORP_ROOT/c/e/d/

nohup $PMSE_BIN/samples/categorization/categorization.pl --root\
$PMCORP_ROOT/c/e/d/ --report 3 --cpus 2 --vector\
'frequent=count=200,distance=tanimoto,weight=1,preprocess=1'` &

P_dvf --in $PMCORP_ROOT/c/e/d/runs/0/categorized.sbl --otype graphic\
--out g.svg
```

Various cmds

```
P_dmp --distance euclid --ngrams may my --out STDOUT

P_cqt --in file --action concordance --query 'e' --out STDOUT

P_dvf --in storable --out STDOUT --sort val

P_ici 'P_rer "s{use encoding (qw)?.utf-?8.}{use utf8}g" [%f]'\
`find -L . -name '*.pm'`
```

18.2 Sentence Segmentation

Sentence segmentation is a standard NLP problem. It is a special case of text segmentation, ”sentence segmentation is the problem of dividing a string of written language into its component sentences.” [1]

The goal of the segmentation is to provide reliable detection of sentence boundaries, which can be a non-trivial task in some languages as these boundaries are denoted by characters with ambiguous meaning. PMSE uses Perls extended regular expression(s) to find these sentence boundaries. By default a newline is used as quickly recognizable/parseable sentence delimiter (one sentence per line). As the whole process is parametrizable, other forms of reorganization are possible.

Before any segmentation can occur, the text should be well prepared and UTF-8 encoded. Wrong punctuation and letter casing may lead to confusing results. `P_trt` (15) can be used to fix some of these problems beforehand.

[1] http://en.wikipedia.org/wiki/Text_segmentation

18.2.1 Basic Segmenter

Here is an example of a short Finnish text:

```
1 - 5- vuotiaat lapset: 2,5 ml oraaliliuosta (puolet 5 ml:n
lusikallisesta) kerran päivässä. 6 - 11- vuotiaat lapset: 5 ml
oraaliliuosta (yksi 5 ml:n lusikallinen) kerran päivässä.
Aikuiset ja yli 12-vuotiaat: 10 ml oraaliliuosta (kaksi
5 ml:n lusikallista) kerran päivässä.
```

The text is formatted in some random way and our goal is to get one sentence per line:

```
1 - 5- vuotiaat lapset: ... kerran päivässä.
6 - 11- vuotiaat lapset: ... kerran päivässä.
- Aikuiset ja yli 12-vuotiaat: ... kerran päivässä.
```

We use `P_trt` to deformat the original text first, which means, the text will loose all formatting information like paragraphs, line breaks etc.

```
$ P_trt --in file.txt --action deformat --out deformatted_text
```

Now the text will have a form of one long line. `P_rer` may be used to break it at the end of sentences. We need to pass a Perl regular expression with the *s* operator, because we need to insert a specific place of the line with line break.

The base of our regexp consists of a terminal punctuation mark: `\p{STerm}`. *STerm* is a Sentence Terminal punctuation. Now follows white-space character and upper case letter:

```
qr{\p{STerm}\s+?\p{Upper}}
```

The beginning of the second and third sentence would not be matched properly. Second sentence starts with a number, third with a dash. Thus we have to add:

```
qr{\p{STerm}\s+?\p{P}?\s*?(\p{Upper}|\d)}
```

Now we have to integrate the regexp with `s{}{}` construction. We will use a look ahead `(?=)` construction to tell the search engine exactly when to match the STerm character. The white-space character will be replaced with a new-line character.

```
s{(\p{STerm})\s+?(?=\p{P}?\s*?(\p{Upper}|\d))}{$1\n}xmsg
```

Most scripts of the world don't have cased letters, e.g. Tamil. In such case you may replace `\p{Upper}` with `\w`.

```
s{(\p{STerm})\s+?(?=\p{P}?\s*?(\w|\d))}{$1\n}xmsg
```

In some scripts (e.g. in Chinese or Japanese), whitespace does not occur between the terminal punctuation mark and the new sentence. Also, the full stop/period is a special character, namely U+3002. We can take advantage of this and apply a special rule.

```
s{(\N{U+3002})}{$1\n}xmsg  # very basic
```

18.2.2 Complex Segmentator

We can have a text with combination of punctuation marks. Or we just want a more robust segmentator:

```
s{(\p{STerm})(?(?<=\N{U+3002})(.))}{$1\n}xmsg
```

We use condition combined with ”look behind” construction. The meaning is ”match any character, if the STerm char is U+3002”. This regexp can be extended:

```
qr{(\p{STerm})(?(?<=\N{U+3002})(.)|\s+?(\p{P}?\s*?(\p{Upper}|\d)))}
```

It means: ”match any STerm character; if it is the U+3002 character, match any following character, else the following character(s) must match a combination of white-space, punctuation and upper case letter or digit”.

Now we need to integrate the regexp into `s{}{}` construction. We will use named backreferences to identify captured sequences:

```
s{
  (?<TERM>\p{STerm})
  (?(?<=\N{U+3002})(?<SENT>.)|\s+?(?<SENT>\p{P}?\s*?(\p{Upper}|\d)))
}{
  $+{TERM}\n$+{SENT}
}xmsg
```

The environment before the STerm character also affects the segmentation. The full stop character in latin scripts may be combined with digits (to express ordinality)[2] or it is a part of abbreviations. We don’t want to match the STerm character in this context, thus we will use ”negative lookbehind” condition in the regexp.

```
qr{(?<!\d)\p{STerm}}
```

Will exclude the context of the STerm character standing after a digit. The negative lookbehind construction can be extended with other non-wanted contexts:

```
qr{(?<!\d)(?<![jJmM][rs]s?)(?<![aA]bbr)\p{STerm}}
```

You can specify as much abbreviations as you want. The complexity of the `s{}{}` construction will grow:

```
s{
  (?<!\d)(?<![jJmM][rs]s?)(?<![aA]bbr)
  (?<TERM>\p{STerm})
  (?(?<=\N{U+3002})(?<SENT>.)|\s+?(?<SENT>\p{P}?\s*?(\p{Upper}|\d)))
}{
  $+{TERM}\n$+{SENT}
}xmsg
```

18.2.3 Advanced Segmenter for Czech

The regular expression grows with the list of abbreviations you want to apply. In such a situation, it is handy to use an INI file for `P_rer`, because you will avoid troubles with the CLI.

You can find an example of such an INI file below. Please note that the *i* section is formatted by newlines to improve readability of this document. The *i* section must be formatted as

[2] This occurs in several European languages: in Croatian, Czech, Danish, Estonian, Faroese, German, Hungarian, Icelandic, Latvian, Norwegian, Polish, Slovak, Slovene, Serbian, Turkish. http://en.wikipedia.org/wiki/Ordinal_indicator

one line in the real INI file.

```
[sentence]
i = (?<!\s(č|f|m|n|p|r|s))
i = (?<!\s(aj|ak|ap|Bc|bl|br|čl|dl|ev|fr|hl|ie|it|kr|kř|
           mj|ml|ms|ob|pf|pl|sg|sl|tj))
i = (?<!\s(abl|adj|adm|adv|akt|arg|atd|atp|att|bás|BcA|boh|
           bot|býv|CSc|csl|dán|dat|děj|
           dep|des|dět|DiS|doc|dol|dop|dór|fam|fem|fil|fin|
           fot|fut|fyz|gen|hod|hor|hud|hut|imp|ind|inf|Ing|
           ión|jap|kpt|lat|lék|les|lid|lit|log|lok|mat|MgA|
           Mgr|mjr|mld|mod|náb|nám|něm|než|niz|nom|nor|odd|
           odp|opt|pas|plk|pol|por|rak|reg|rkp|rtm|rtn|rum|
           rus))
i = (?<!\s(alch|amer|anat|angl|apod|arab|arch|astr|belg|bibl|biol|
           brit|bulh|církˇ|dial|dopr|dosl|ekon|epic|film|form|geol|
           geom|germ|gram|hebr|hist|horn|chem|chil|impf|iron|JUDr|
           klas|kniž|komp|konj|kuch|metr|MUDr|MVDr|mysl|např|npor|
           nrtm|obch|obyč|odst|ojed|part|pers|PhDr|plpf|pomn|popř|
           pplk|ppor|prap|práv|prep|prof|rcsl|refl|resp|RNDr|RSDr|
           slov))
i = (?<!\s(absol|eufem|event|geogr|hovor|chcsl|instr|kanad|konkr|
           námoř|nprap|pejor|pprap|předl|přivl|slang|s\.r\.o))
i = (?<!\s(archit|astrol|genmjr|genplk|genpor|herald|interj|liturg|
           meteor|neklas|nstržm|samohl))
i = (?<!\s(etnonym|indoevr|katalán|nesklon|PharmDr))
i = (?<!\s(anglosas))
i = (?<sep>\p{STerm}\p{QMark}?\s?)\s*?(?=\p{QMark}?\s?\p{Upper})
o = $+{sep}\n
m = xmsg
```

Resulting regexp takes advantage of multiple negative look-behind conditions. Each of these concatenates abbreviations (in 'OR' relation) of the same length.

Abbreviations of titles[3] and abbreviations of textual units e.g.: *par. 6* or *par. F* would cause false boundary match.

The INI file may be applied with P_rer like this:

```
$ P_rer --bulk ssegmentation.ini text.txt
```

The INI file mentioned above was created by `buildrx_neglookbehind_from_abbrevs`[4] - a help script that builds complex INI files from list of abbreviations. Please note that you may use multiple `i` and also `o` sub-sections for P_rer's bulk file. Also, you may use multiple sections. `P_rer` has option `--sect` for specification of particular section.

Abbreviations are placed in `$PMSE_ROOT/cfg/P_rer/abbreviations`. You will find there abbreviations for all major languages in EU. Resulting INI files are stored in `$PMSE_ROOT/cfg/P_rer/segmenter`.

[3] E.g.: titles stand often before the names in the Czech language, e.g.: PhDr. Jan Novák.
[4] This script is placed in `$PMSE_DEVBINdirectory`.

18.3 Sub Word N-grams Extraction

In this manual, a *sub word n-gram* is a string of characters smaller than a word. subword N-grams are capable to describe repetitive patterns of graphemes. Formally, a subword n-gram is just a substring of the input text.

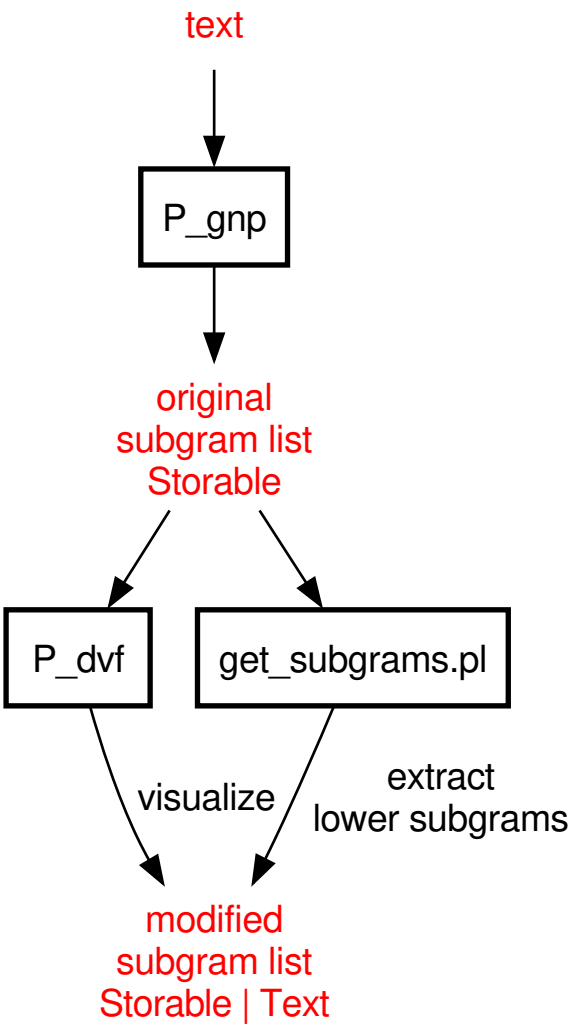

Figure 18.1: Schematic overview of subgrams extraction

In other words: it is an n-gram, but the units, from which the subword n-gram consists, are not word-like tokens, but graphemes. E.g.: the word *grapheme* consists of these subword n-grams of size 2:

```
gr ra ap ph   # inside a word
he em me

_g            # over the word boundary
e_
```

The subword n-gram may occur in several positions: inside the word, on the word boundary and over the word boundary. If the length of the subword n-gram is large, it may consists of several words. We will mark the white-spaces, in order to identify (cross) boundary

subword n-gram.

First of all, deformat the text to remove indention and multiple white-spaces.

```
$ P_trt --in text.txt --action deformat --out deformatted
```

Now apply P_rer on the deformatted text. We want to mark white-spaces

```
$ P_rer 's{\s}{_}xmsg' deformatted/repaired
```

and remove the punctuation:

```
$ P_rer 's{\p{P}}{}xmsg' deformatted/repaired
```

Now all the white-spaces transformed into the underscores. (The marker is - as usual - arbitrary. You can use white-spaces as well.) Now we need to insert underscores in the beginning and the end of the text. You should insert ”length of the subword n-gram” - 1 of markers. If the length of the subword n-gram would be 4, you need to insert 3 markers in the beginning and 3 in the end.

```
$ P_rer 's{\A(.*)\z}{___$1___}xmsg' deformatted/repaired
```

The text is prepared for processing. We will use P_gnp:

```
$ P_gnp --in deformatted/repaired --out subgrams --delimiter '' \
>  --cluster count  --ngram 4 4 ''
```

The resulting Storable file will be stored as `subgrams/count`. The subword n-grams of length 4 allow you to distinguish the minimal (length 2) subword n-grams in all positions. Consider a subword n-gram ’of’. It can occur as a self standing word (preposition), on the word boundary, or inside the word:

```
_of_        # preposition
eof_        # word boundary (thereof)
rofi        # inside word (profile)
```

You can select/filter the subword n-grams with P_dvf (filter option), or set ’ofilter’ option in P_gnp. We will use P_dvf to get subword n-grams occuring **in** the boundaries of a word:

```
$ P_dvf --in subgrams/count --filter '$key =~ m{_}xms' \
>   --out subgrams.sbl --otype storable
```

The resulting file `subgrams.sbl` (in Storable format) consists still of subword n-grams of length 4. To extract subword n-grams of length 2, use auxiliary script `get_subgrams.pl`, which is placed at `$PMSE_BIN/samples/get_subgrams.pl`.

Idea behind the script is easy: provide a regular expression, which will be applied on each subword n-gram of the list. You can match only the type of subword n-grams you want to work with. (The filtering step with P_dvf is not necessary.) The original subword n-gram will be re-created and its frequency re-counted. The output is the new subword n-gram list with re-counted frequencies stored in a Storable file. The subword n-gram list may be printed on STDOUT.

```
# for filtered subgram list:
```

```
./get_subgrams.pl --in subgrams.sbl --regexp '\w(?<sgram>\w{2})\w'

# for non-filtered subgram list:

./get\_subgrams.pl --in subgrams.sbl \
--regexp '\A[^_](?<sgram>[^_]{2})[^_]\z'}
```

The mentioned command line example(s) will print in-word n-grams of length 2 (and their counts) on STDOUT. Note: While generating the original subgram list, it is important to specify `--cluster count` option (in P_gnp). Only basic counts of subword n-grams may be re-counted.

If you want to get the the probality of subword n-grams, specify `--action prob` when calling `get_subgrams.pl`.

18.4 Probability of Neighbors

This recipe will show how to extract probability of occurrence of tokens on nearest contextual positions. Assume following text[5]:

He was the third child of eight and the eldest surviving son. Shakespeare produced most of his known work between 1589 and 1613. Many of his plays were published in editions of varying quality and accuracy during his lifetime. In 1623, John Heminges and Henry Condell, two friends and fellow actors of Shakespeare, published the First Folio, a collected edition of his dramatic works that included all but two of the plays now recognised as Shakespeare's. Shakespeare was buried in the chancel of the Holy Trinity Church two days after his death.

Now we would like to know probability of tokens occurring on the first position after preposition *of*, which has 8 occurrences in the sample text. There exist namely these pairs:

```
of eight
of his
of his
of varying
of Shakespeare
of his
of the
of the
```

The probability of co-occurrence *of eight*, is 1/8, prob. of *of his* is 3/8, *of the* is 2/8 etc. PMSE has a script called `get_context_probability` that is intended exactly for this operation. The script is placed in `$PMSE_DEVBIN`. It needs two input files: file storing count of bigrams and file storing probability of unigrams (generated from the same source as the file with bigrams). Both input files should be generated with P_gnp, see chapter 11. Output is a data-structure where each unigram has a predecessor (marked as `°+1`) and successor (marked as `°-1`) position. Count of tokens on each position may be also set (default is 5) as well as output limit of "root" unigrams (default is 100,000).

```
of => {                          # root unigram
```

[5] It is a sample of an article about William Shakespeare published on Wikipedia: http://en.wikipedia.org/wiki/Shakespeare

```
    P   => 0.0860215053763441,       # probability of root unigram
    °+1 => {                         # predecessors
         Many    => { P => 0.125 },
         actors  => { P => 0.125 },
         child   => { P => 0.125 },
         edition => { P => 0.125 },
         most    => { P => 0.125 },
    },
    °-1 => {                         # successors
         Shakespeare => { P => 0.125 },
         eight       => { P => 0.125 },
         his         => { P => 0.375 },
         the         => { P => 0.25  },
         varying     => { P => 0.125 },
    },
}
```

Help for script `get_context_probability` may be invoked as in other PMSE scripts:

```
$ perl get_context_probability -?
```

Output (data structure similar as listed above) is stored in Storable format, it is PMSE::Visualize::Neighbors object and has some predefined method of conversion, currently to text and YAML. You can do that with P_dvf, see chapter: 10.

18.5 Co-occurrences

Our concept of co-occurrences is similar to kocos.pl in Text::NSP[6]:

> Co-occurrences are the words which occur together in the same context. All words which co-occur with a given target word are called its co-occurrences. The concept of 2nd order co-occurrences is explained in the paper Automatic word Sense Discrimination [Schutze98]. According to this paper, the words which co-occur with the co-occurring words of a target word are called as the 2nd order co-occurrences of that word.

The words which co-occur with the 2nd order co-occurring words belong to 3rd order and so on.

18.5.1 What is a Co-occurrence in Linguistics?

The linguistic term of *co-occurrence* is related to the term of *collocation*. Generally speaking, collocation and also co-occurrence provide information about context of a word, or - by collocation - about multi-word units. The strength of the collocation is derived from the probabilities of occurrence its components. The measures of collocational strength take into account the mutual position of the lexical units and the separate occurrence of the units in the corpus.

[6] Available from http://search.cpan.org/dist/Text-NSP/lib/Text/NSP.pm

The concept of a co-occurrence is different. It describes the relation of a word to other lexical units but without a dependency on absolute position of the word in the text.

If we have e.g. following sentences from which we want to extract basic bigrams (n-grams of size 2 from window 2):

```
I like Chinese language. It is a rich language.


I like
like Chinese
Chinese language
language It
It is
is a
a rich
rich language
```

Then co-occurring words for *Chinese* are:

```
Chinese   target

like      1st order
language  1st order

rich      2nd order
```

There exists dependency chain among *Chinese - language - rich*. Note that *Chinese* and *rich* stay far from each other in the text.

18.5.2 Extract Co-occurrences

Then we will use P_cqt to extract the co-occurrences and then we will visualize them with P_dvf.

We need to extract bigrams first, thus we will use P_gnp. Let the the input file be 'source.txt':

```
$ P_gnp --in source.txt --cluster count --out bigrams --ifilter '-token=\s+'
```

Now, do some filtering[7] and store the bigrams as non-PMSE object:

```
$ P_dvf --in bigrams/count --out bigrams.sbl --otype storable \
>   --filter '$value < 100 || $key =~ m\{\p{P}}xms'
```

Finally, extract the co-occurrences with P_cop:

```
$ P_cop --in bigrams.sbl --level 2 --target 'literal=Chinese' --out coocs
```

And display the result with P_dvf:

```
$ P_dvf --in coocs/cooccurrences --otype graphic --out cooc
```

[7]The filtering step is necessary, if you want to do a qualitative inspection of interesting relations among words, you will need to filter out the bigrams containing punctuation and probably grammatical words. Also, you will need to filter out bigrams with low frequencies, because they would cause the graphic output to be poorly arranged.

P_dvf should create a file called `cooc.svg`, which should look like picture 18.5.2.

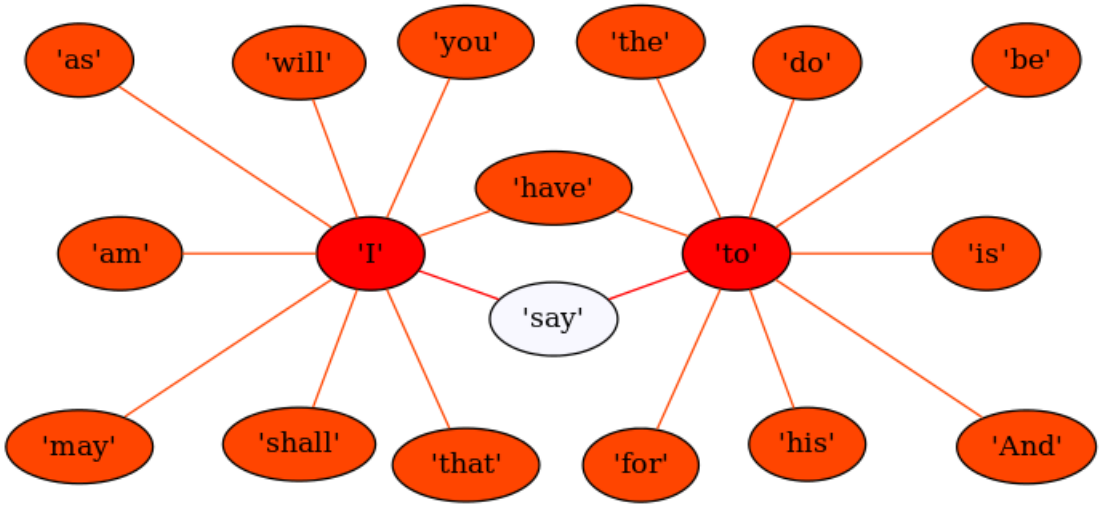

Figure 18.2: The Canterbury Tales, and Other Poems by Geoffrey Chaucer - grammatical co-occurrences for 'say' (filtered input)

The target (*say*) is marked with white color. Co-occurrences of first order are marked by red, co-occurrences of second order are marked by orrange colour. The word *have* is connected with two words, because it co-occurs with both of them.

18.5.3 Convert Text::NSP Bigrams to PMSE

Text::NSP use following format to store bigrams:

```
11
line<>of<>2 3 2
of<>text<>2 2 2
second<>line<>1 1 3
line<>and<>1 3 1
and<>a<>1 1 1
a<>third<>1 1 1
first<>line<>1 1 3
third<>line<>1 1 3
text<>second<>1 1 1
```

This list can be stored e.g. in a file called 'nsp.txt'. We need to get a format which we can load in P_dvf. Thus we will use P_rer to convert the NSP list:

```
P_rer 's{^(.+?)<>(.+?)<>(\d+?)\s.+?$}{$1<>$2\t$3}xmsg' nsp.txt
```

We get a list like this one:

```
11
line<>of 2
of<>text 2
second<>line 1
line<>and 1
and<>a 1
```

```
a<>third 1
first<>line 1
third<>line 1
text<>second 1
```

The last thing we need to remove is the total count of bigrams at the beginning of the file:

```
P_rer 's{^\d+?\n}{}xms' nsp.txt
```

Now we can load the list into P_dvf:

```
$ P_dvf --in nsp.txt --itype text --otype storable
```

The result is a histogram-like structure in the Storable format.

18.6 Text Categorization

The categorization of texts (TextCat) is a use-case for PMSE. It was intended as a simply layer of glue connecting parts of the environment first. However, it has developed in the regular procedure which integrates several parts of PMSE as well as several modules in the libraries.

Great attention was paid for efficiency as well for modularity of the system. [8]

18.6.1 Brief Description of the Procedure

First of all, let us consider a directory structure as is described in the section 2.2. The first step is to convert the source files into a plain text. Consider a categorization of Czech text.

The path to the texts will be: `$PMCORP_ROOT/c/e/s/original`. The conversion is a task for P_dmf:[9]

```
$ P_dmf --in /data/library/c/e/s/original
```

When we have created the *derived* directory and the text files, we can run the categorization.

```
$ $PMSE_BIN/samples/categorization/categorization.pl \
>   --root $PMCORP_ROOT/c/e/s  --report 3 --cpus 2 \
>   --vector frequent = 'count=200,distance=tanimoto,weight=1,preprocess=1'
```

If the categorization process was completed successfully, we should see directory called runs. In this directory are stored runs of the TextCat. Each run has own directory called by number, first run of categorization will be stored in directory 0. Input options for each run are reported in *categorization.env* file, which is included in all run directories. You can display the file with:

```
$ P_dvf --in categorization.env
```

Now go into the `runs/0` directory and list all files. You should see a file called *categorized.sbl*. Load this file into P_dvf to visualize it. This file contains the result of categorization run.

[8] Instructions how to write a TextCat module are placed in the PMLS manual.
[9] You have to use the absolute path.

It is a Storable file containing a binary tree: the structure of clusters based on the lexical proximity of the input texts.

Command below will create a file *g.svg* in the SVG format.

```
$ P_dvf --in categorized.sbl --otype graphic --out g.svg
```

To get detailed info about the output data structure, see section 18.7.3.

18.6.2 Categorization.pl - Interface for TextCat

The script *categorization.pl* is rather a wrapper of various PMSE functions than a regular PMSE script. Therefore it is placed in `$PMSE_BIN/samples/categorization` directory and does not have a regular PMSE name. The function of the script is to provide a simply interface for the process of textual categorization.

The script has several options:

categorization.pl Categorize texts

SYNOPSIS

```
categorization.pl OPTIONS
```

OPTIONS

–analyze

Count Entropy and Purity measure for the clusters of the graph.

This assumes you know the categories of texts BEFORE categorization. To get correct results of cluster evaluation, you have to name your input texts correctly. This function is designed for text-names like:

```
perl-1.txt
```

Where 'perl' is a name of the category. Only this string will be taken as a category name. The categories are recognized automatically from the names of the texts. The dash '-' and '.txt' suffix are mandatory.

–cpus <**n**>

Number of threads that should be using during categorizatino process. 0 means no fork.

–filter <**hash**>

Defines filtering tokens/files. Default no filtering.

```
files=pareto
tokens=pareto
```

–report <**level**>

Reporting level used in PMSE scripts. Levels could be usually 1, 2 or 3.

–**root** <**directory**>

Directory is of the form

```
*/./././
```

and it contains derived directory. Inside there is a file structure with .txt files representing corpora. This structure could be created by `P_dmf`. By default /data/library/m/u/l/.

–**vector** <**hash**>

Defines categorization criteria. For most common `frequent` choice we have following parameters:

```
count       number of frequent ngrams
distance    name of distance measure
preprocess  what should be considered: 1 for unigrams etc.
```

We have also few universal parameters:

```
weight      importance of this criterion
```

Default frequent='count=200,weight=1,distance=tanimoto,preprocess=1'.

Example with 2dimensional vector:

```
-vector frequent='weight=0.8,count=200,preprocess=1,\
distance=tanimoto'

-vector slength='weight=0.2,...'
```

Note: combinations of more `--vector` criteria is not implemented.

18.7 PMSE Visualization

Main visualization tool of the PMSE framework is the P_dvf script. However, the process of visualization begins earlier, when the particular data-structure is being created. Each data-structure created in the PMSE framework is an *object* which may have a specific set of properties or methods.

18.7.1 Objects In PMSE

PMSE objects have pre-defined methods for visualization and format conversion. [10] This is handy, because the input data structure may be loaded in the P_dvf script and there will be the data structure automatically converted or visualized according to predefined methods (and input options).

The objects share several common visualization methods, but even so - the difference in the structures affects the attributes and options of these methods. This section of cookbook is intended to provide detailed info about the objects and their methods specifics - visualizing options.

[10] In the current version, PMSE stores objects information both about visualization and conversion to other formats. This may be changed in the future.

18.7.2 Input from the Outer Space

The visualization of data created by other applications is possible. PMSE will load in the data structure and will create a basic object with five basic methods of visualization:

- printed data structure P_dvf –otype pdump
- storable P_dvf –otype storable
- text P_dvf (text is default otype option)
- yaml P_dvf –otype yaml
- SVG graphic P_dvf –otype graphic

These methods are default for all PMSE objects, however - the complexity of the graphical output is slightly different. The basic PMSE object is a simply hash with a `data` section which is printed out in few possible formats.

18.7.3 Binary Tree Visualization

Binary tree is a common data structure; in case of PMSE, it is formed by categorization process - see section 18.6 for details. The output structure consists of clusters - groups of texts which represent the similarity among texts.

PMSE uses the GraphViz tool to visualize this structure. Now consider a simple example. We performed a categorization of 8 texts. The input data stored in `original` directory was like this:

```
original/
  --text/
     --Dinoland-0.txt
     --Dinoland-1.txt
     --MarkStone-2.txt
     --MarkStone-3.txt
     --PerryRhodan-4.txt
     --PerryRhodan-5.txt
     --RenDhark-6.txt
     --RenDhark-7.txt
```

After we ran `categorization.pl` script, we should see a directory `runs/<number of run>`. In this directory should be place file called `categorized.sbl`. We can display it simply as:

```
$ P_dvf --in categorized.sbl
```

What shoud give us something like:

```
{
  0 "text/PerryRhodan-5.txt/lvl.last/PerryRhodan-5.txt/PerryRhodan-5.txt",
  1 "text/PerryRhodan-5.txt/lvl.last/MarkStone-3.txt/MarkStone-3.txt",
  2 "text/PerryRhodan-5.txt/lvl.last/MarkStone-2.txt/MarkStone-2.txt",
  3 "text/PerryRhodan-5.txt/lvl.last/RenDhark-6.txt/RenDhark-6.txt",
  4 "text/PerryRhodan-5.txt/lvl.last/Dinoland-0.txt/Dinoland-0.txt",
  5 "text/PerryRhodan-5.txt/lvl.last/PerryRhodan-4.txt/PerryRhodan-4.txt",
  6 "text/PerryRhodan-5.txt/lvl.last/Dinoland-1.txt/Dinoland-1.txt",
  7 "text/PerryRhodan-5.txt/lvl.last/RenDhark-7.txt/RenDhark-7.txt",
```

```
  8 [
        [0] 6,
        [1] 4
    ],
  9 [
        [0] 7,
        [1] 3
    ],
  10 [
        [0] 2,
        [1] 1
    ],
  11 [
        [0] 5,
        [1] 0
    ],
  12 [
        [0] 11,
        [1] 9
    ],
  13 [
        [0] 12,
        [1] 10
    ],
  14 [
        [0] 13,
        [1] 8
    ]
}
```

The numbers in the first column express the clusters. The lowest clusters 0 - 7 consist of ”leafs” - single texts which are combined in higher clusters until the last super-cluster (root) - 14 is reached. The command for forming of the graphical output is:[11]

```
$ P_dvf --in categorized.sbl --otype graphic --out g.svg
```

Ii will give us a graph like:

The look of the graph is specified in PMSE::Visualize::BinaryTree module. There exist two additional features of the graph:

First is the analysis of the cluster(s). If the `--analyze` option in `categorization.pl` is in use, values for entropy and purity measure will be counted. These measures are used to evaluate the ”quality” of the cluster(s) - both measures take in account the ratio of categories contained in the cluster. The ideal value of purity is 1, entropy - on the opposite - has ideal value 0. You have to know the distribution of texts in given categories before you start the categorization.

[11] The SVG output format is available only from Perl version 5.14.2 and higher. If the user runs lower version of Perl, the output will be stored in a `dot` format. For explanation of `dot` see e.g.: http://en.wikipedia.org/wiki/DOT_%28graph_description_language%29

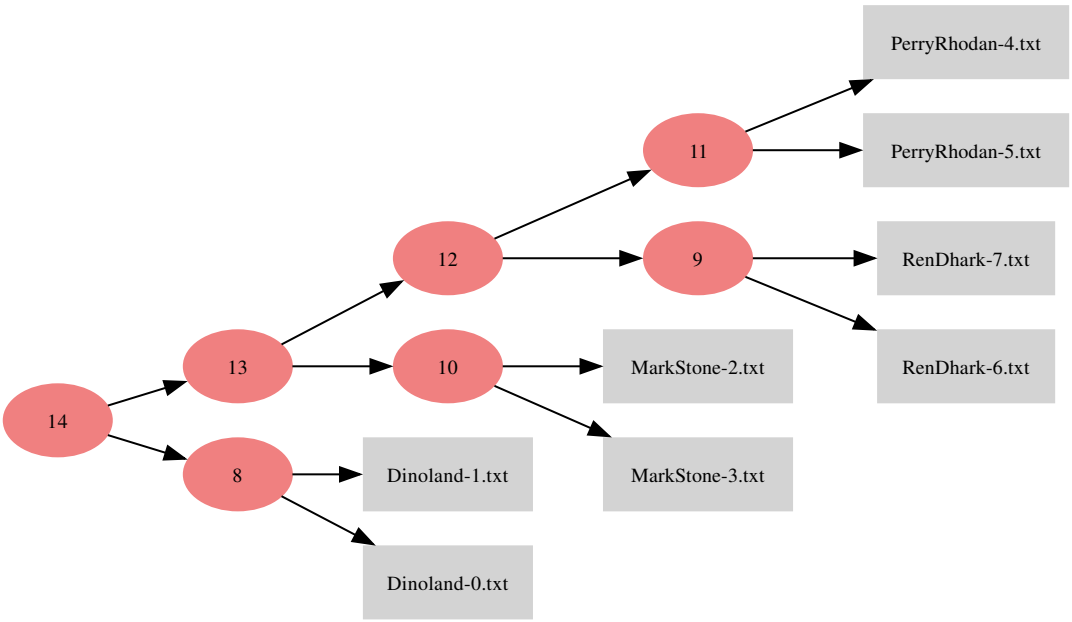

Figure 18.3: Example of clusters visualized by GraphViz

In order to count entropy and purity, it is necessary to use appropriate names of the input texts. The name of text should contain number of category to which the text belongs, a hyphen, number and the `.txt` suffix. The example from this section shows a group of 8 texts divided into 4 categories (4 science-fiction series). Each category consists of 2 (pulp-booklet) novellas belonging to a specific series.

Second the box-like nodes which contain the name of text should contain a link (a path to the file in your file system) aiming on the file. If you open the SVG graphic in a web browser, the name of the text should be clickable. You can easily inspect the content of the text files.

18.8 Visualization of Contingency Tables

Contingency table is another type of output created by P_gnp. Contingency tables in PMSE store information about the n-grams generated from given input file. The table stores information about frequency of occurrence of tokens (and their possible combinations) from which is the n-gram comprised. In the case we process n-grams with n bigger than 2, the tables are n-dimensional.

```
$ P_gnp --in <some_text> --contingency --out cont --ifilter '+token=\w+'
```

P_gnp stores the contingency table as a Storable file.

18.8.1 Interpretation of the Data Structure

Consider we have following sentence: *I am home here.* If we generate bigrams from this sentence, we will get a hash like this:

```
my $contingency = {
```

```
    'I am'      => [1, 0, 0, 2],
    'am home'   => [1, 0, 0, 2],
    'home here' => [1, 0, 0, 2],
};
```

The left side (keys of the hash) of the table holds the particular n-gram. The right side (values of the hash) contains an array filled with values of occurrences of combinations of tokens from the n-gram.

The combination of tokens is encoded as a binary number in the position of the value in the array. For example the first row of the table is:

```
    'I am'      => [1, 0, 0, 2],
```

The first value from the array has in Perl index 0. Decimal 0 is in binary also 0 and 0 denotes the true existence of the token in the n-gram. The first value in the array therefore denotes, that the the combination *I am* occurs exactly once in the input text.[12]

The second number in the array: it has index 1. We can interpret it as 01[13] in the binary form. This denotes combination of *I* with any other possible token (except *am*) from the input text - we see, that there is no such combination in the input text, therefore it has value 0.

The third number in the array has index 2 - in binary form 10. This combination denotes combination of any token (except*I*) and *am*. We see no *non-I am* combination exists in the input text.

The fourth number has index 3 - in the binary form 11. This means any combination of *no-I* and *non-am* in the input text. We see we have two such combinations: *am home* and *home here*, therefore the value is 2.

This encryption of combinations is universal for n-grams of any length.

18.8.2 Methods of Visualization

in P_dvf:

As Text / CSV The basic method is as a plain text. No output format specification is needed, because the plain text is a default output format of P_dvf. In the case of contingency tables, the output format is rather a CSV, because it is more practical.

```
$ P_dvf --in cont/contingency --out cont.csv
```

The output will look like the following:

```
,home,not home
am,1,0
not am,0,2
,here,not here
home,1,0
not home,0,2
```

[12] The binary 0 could be interpreted as 00 - TOKEN TOKEN.

[13] TOKEN - NONTOKEN. NONTOKEN means, that we assume any other token except the real one existing on given position.

```
,am,not am
I,1,0
not I,0,2
```

Now you can load cont.csv e.g. into a spread-sheet editor or R.

SVG Graphic To visualize a contingency table, we use R with the combination of **vcd** package.[14] We use a mosaic plot (in SVG format) for the visualization, however this functionality is a little bit experimental for now. The command to get the graphic is:

```
$ P_dvf --in cont.sbl --out cont.svg --otype graphic
```

The readability of the graph depends strongly on the count of rows in the contingency table. If there is a lot of them, the graph won't be readable. Therefore you may need to filter them.

The filtering is specified via a Perl code - you can specify the keys (n-grams) and values (the real frequency of occurrence of the given n-gram) you want to remove from the table.

```
$ P_dvf --in cont.sbl --out cont.svg --otype graphic --filter \
>   '$key !~ m{\bI\b}'
```

The picture 18.8.2 shows three most frequent bigrams extracted from Chaucer's Canterbury Tales. Each colored box of a row shows the proportion of frequency of occurrence for given combination of tokens.

18.8.3 Distance Visualization

The input data structure is a nested hash consisting of a name of given distance measure(s), info about normalization, compared n-grams and the value(s) of given distance measure(s).

The basic methods of visualization are `spreadsheet` and `text`.

```
$ P_dvf --in dir/distance --otype spreadsheet --out distance.csv
```

18.8.4 FileStat Visualization

The input data structure is a simply hash with statistical information related to a specified text. The basic methods of visualization are `spreadsheet` and `text`.

```
$ P_dvf --in dir/overview --otype spreadsheet --out overview.csv
```

18.9 N-grams Histogram Visualization

Consider a following data structure:

```
...
the<>language   3
language<>is   3
```

[14] http://cran.r-project.org/web/packages/vcd/index.html

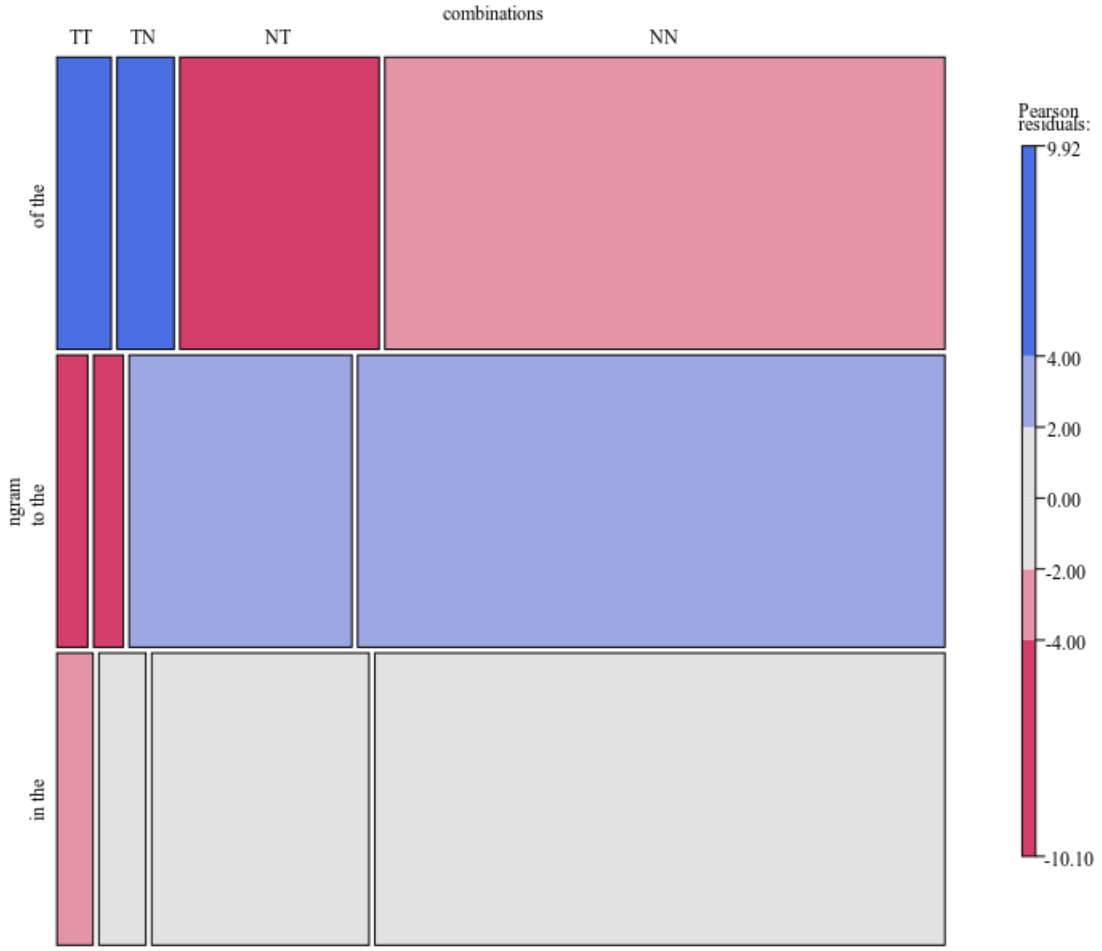

Figure 18.4: The Canterbury Tales, and Other Poems by Geoffrey Chaucer - three most frequent bigrams

```
to<>use    3
agree<>on    3
programming<>in    3
ten<>percent    2
one<>can    2
which<>ten    2
...
```

These are n-grams generated by P_gnp, e.g.:

```
$ P_gnp --in text.txt --cluster count --ngram 2 2 '<>' \
> --ifilter '+token=\w+' --out try
```

And visualized with P_dvf:

```
$ P_dvf --in try/prob --sort '+val'
```

P_dvf offers multiple output formats, 'graphic' among others. Word list or a N-gram list has a predefined graphic format - a word cloud. Thus when you give following command:[15]

```
$ P_dvf --in try/count --otype graphic --out graph
```

You will get an svg file called 'graph.svg', similar to this:

18.9.1 Histogram Visualization Commands

Histogram has several specific methods - options:

- bulk: building of more complex regexp for filtering, e.g.: stop-lists
- filter: filter keys or values
- fmt: specify key - value position on the line
- limit: limit the number of lines with output
- sort: sort keys or values

Exampples of commands for P_dvf related to histogram:

```
P_dvf --otype pdump                          # printed data
                                             # structure
P_dvf --sort <sort_order>  --fmt <fmt>       # text
P_dvf --sort <sort_order>  --otype yaml      # yaml
P_dvf --filter <code>
P_dvf --otype graphic                        # svg
P_dvf --bulk <ini file>
```

Here follows short description of histogram-specific options:

–bulk <file>

The bulk specifies filtering options. It is handy for specifying of stop-lists.

Define a bulk file with code that will be applied during filtering. We can achieve multiple filtering options with this. There must be defined at least one [section] called 'eval' and hook called 'filter':

```
[eval]
filter = <<END
<code>
<code>
END
```

We can use heredoc style to define multiple code blocks. To specify the code, look at the option `--filter`.

–filter <code>

[15] This is an early implementation only. If you want to use the 'graphic' output format, you have to generate the input file with P_gnp –object option.

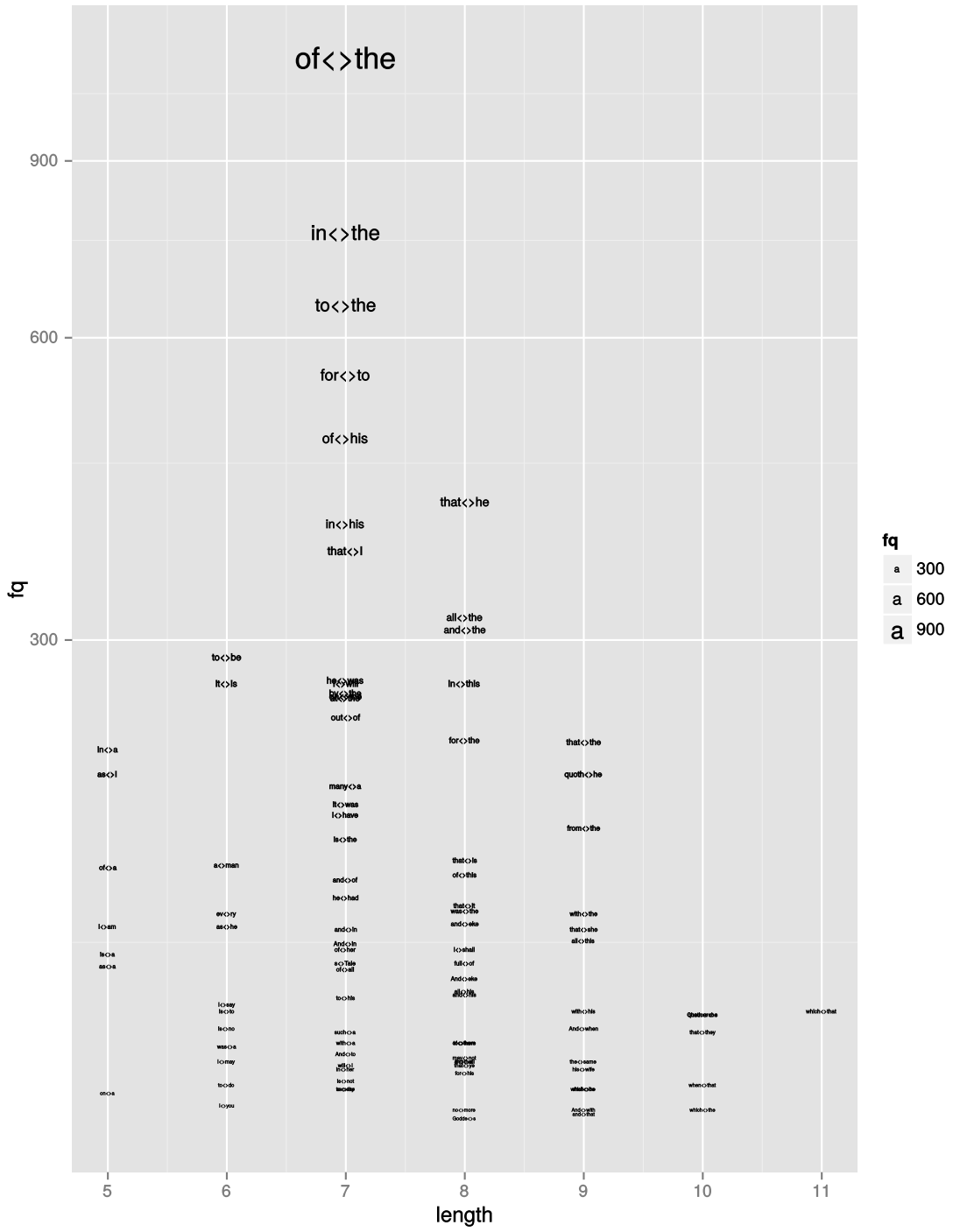

Figure 18.5: The Canterbury Tales, and Other Poems by Geoffrey Chauce: most frequent bigrams

Filter is an optional choice independent on the output format.

Filtering is done via perl code. As all output data is stored in a hash (list of 'key-value' pairs), the user can define filtering expressions like:

```
$key   =~ <regex>
```

```
$value > <value>
```

Data which matches the regex, or pass the comparsion, are deleted.

–fmt <template>

Fmt is optional - available only for text (default) output format.

Format string to define the output. The variables `%k` and `%v` denote positions of the current key and value. If not given, the default is `%k \t %v\n`.

–limit <number>

Limit is optional - available only for text (default) output format. Limit the number of output lines to C<num> (integer). The default is 0, in which case the output is unlimited. If this parameter is defined just the first C<num> lines are printed.

–sort <sort_order>

Sort is available for otype(s):

```
as_printed_data_structure
as_yaml
as_text (default)
```

Sort is optional. Valid values for <type> are:

```
+key  sort by key in ascending order (default)
-key  sort by key in descending order (default)
+val  sort by value in ascending order
-val  sort by value in descending order
```

18.10 Macros

The idea of a PMSE macro is to provide a predefined sequence (chain) of PMSE commands with already specified input parameters. Macros are often built using shell scripts, but basically every formalism allowing you to chain shell commands will do. Macros are handy to build larger building blocks of functionality from lower level commands.

18.10.1 MAK_1s1l

Consider the application of sentence segmentation for different languages. We describe in the section 18.2, how to write a sentence segmentator. In the core of this functionality, `P_rer` is used with language-specific regular expressions. In such case it would be necessary to have different bulk files for files in different languages.

The example below shows a simple macro called *MAK_1s1l* which is basically a simple if - elsif branch:

```
#!/bin/sh
#
# PMSE MAKRO: execute sentence segmentation for a file with
# language specific segmentator. First, the script creates
# a copy of the input file with 'ss-' prefix.
# The copy is segmented in the next step.
#
```

```
FILE=$1   # input 1
LANGUAGE=$2   # input 2

cp $FILE "ss-$FILE"

# lang CES

if [ "$LANGUAGE" = "ces" ] ; then

P_rer 's{(?<!angl)(?<!\smj)(?<!\sIng)(?<!\sBc)(?<!\sbelg)
         (?<!\sbiol)(?<!\sč)(?<!\sčl)(?<!dán)(?<!\sfyz)
         (?<!\sgenmjr)(?<!\sgenpor)(?<!Ing)(?<!\sJUDr)(?<!\smat)
         (?<!\sMgA)(?<!\sMgr)(?<!\smjr)(?<!\sml)(?<!\sMUDr)
         (?<!\sMVDr)(?<!\snapř)(?<!\snpor)(?<!\snprap)
         (?<!\sodst)(?<!písm)(?<!\ss)(?<!\sr)(?<!\sspol)
         (?<!\sPharmDr)(?<!\sPhDr)(?<!\splk)(?<!\spopř)
         (?<!\spplk)(?<!\sppor)(?<!\spprap)(?<!\sprap)
         (?<!\sRNDr)(?<!\sRSDr)(?<!\sThDr)(?<!\stzv)(?<!\sjm)
         (?<!\svl\.jm)(?<!\szkr)(?<!\szn)(?<!\szvl)
         (?<STERM>\p{STerm}\p{QMark}?)\s+?(?=\p{P}?\s*?(\p{Upper}|\d))}
         {$+{STERM}\n}xmsg' "ss-$FILE"

# lang ENG

elif [ "$LANGUAGE" = "deu" ] ; then

P_rer 's{(?<!\sAttn)(?<!\sc\.c)(?<!\scf)(?<!\se\.g)(?<!\senc)
         (?<!\sencl)(?<!\sref)(?<!\sf\.o\.a)(?<!\sw)(?<!\si\.e)
         (?<!\sinc)(?<!\sYrs)(?<!\sadmin)(?<!\sh\.q)
         (?<!\sMan\.Dir)(?<!\sa\.o\.b)(?<!\sassoc)(?<!\sdept)
         (?<!\srep)(?<!\sest)(?<!\sexec)(?<!\sext)(?<!\sXer)
         (?<STERM>\p{STerm}\p{QMark}?)\s+?(?=\p{P}?\s*?(\p{Upper}|\d))}
         {$+{STERM}\n}xmsg' "ss-$FILE"

# lang DEU

elif [ "$LANGUAGE" = "eng" ] ; then

P_rer 's{(?<!\sapl)(?<!\sd)(?<!\sDipl)(?<!\sDr)(?<!\sG)
         (?<!\sd\.h)(?<!\sevtl)(?<!\sgepfl)(?<!\si\.H\.v)
         (?<!\ssw)(?<!\su\.v\.a)(?<!\sV\.i\.R)(?<!\sz\.B)
         (?<!\sMon)(?<!\stgl)(?<!\sProf)(?<!\su\.A\.w\.g)
         (?<!\su\.U)(?<!\sv\.a)(?<!\sz\.d\.I)(?<!\sz\.I)
         (?<!\sz\.I\.I)
         (?<STERM>\p{STerm}\p{QMark}?)\s+?(?=\p{P}?\s*?(\p{Upper}|\d))}
         {$+{STERM}\n}xmsg' "ss-$FILE"

else
```

```
echo "Language $LANGUAGE is not known!"

fi
```

`$1` and `$2` are the input variables - they contain a value specified as an input argument on CLI. The first value is a filename and the second - an iso-639-3 code - the language specification. The second variable is matched with predefined strings (also iso codes).

The script will make a copy of the input file (new file with *ss-* prefix will be created). Which will be modified afterwards.

If the specific code matches, the appropriate segmentator is used.[16]

If the second input parameter doesn't match any predefined code, the script will print a warning.

18.10.2 Further extensions

It may be useful to integrate some text processing functions in the macro. The name of the macro is *MAK_1s1l* - one sentence per one line. However, the input text could be wrong formatted, it could contain e.g. multiple empty lines or headlines; the resulting file could be affected as well.

`P_trt` can be used to remove headlines (titles) from the text and to remove all formatting characters (mainly line breaks). Following command should replace the copy `cp $FILE "ss-$FILE"` command:

```
P_trt --action remove_headline --in $FILE --out STDOUT | \
P_trt --in STDIN --action deformat --out STDOUT > "ss-$FILE"
```

Here is a little example[17] of what will happen with the input text. Let the original text be[18]:

```
Language <br>
<br>
Language is the human capacity for acquiring and using complex
systems of communication, and a language is any specific example
of such a system. The scientific study of <br1>
language        is           called linguistics.<br>
<br>
Estimates of the number of languages <br2>
<br3>
in the world vary between 6,000 and 7,000.<br>
```

After the first `P_trt` command the text should look like:

```
Language is the human capacity for acquiring and using complex
systems of communication, and a language is any specific example
of such a system. The scientific study of
language        is           called linguistics.
```

[16] The segmentators above are only examples - they contain abbreviations which may cause a problem in the segmentation. The enumeration of the abbreviations is definitely not final.

[17] From http://en.wikipedia.org/wiki/Language

[18] The <*br*> signs denotes line breaks. They are not a part of the original text - they are used to visualize the problem. We want to remove breaks 1, 2 and 3, because they corrupt the sentences

```
Estimates of the number of languages in the world vary between
6,000 and 7,000.
```

The second `P_trt` command will remove all other formatting, thus the text will have a form of:

```
Language is the human capacity for acquiring and using complex
systems of communication, and a language is any specific example
of such a system. The scientific study of language is called
linguistics. Estimates of the number of languages in the world
vary between 6,000 and 7,000.
```

The final (segmented) text will look like:

```
Language is the human capacity for acquiring and using complex
systems of communication, and a language is any specific example
of such a system.<br>
The scientific study of language is called
linguistics.<br>
Estimates of the number of languages in the world
vary between 6,000 and 7,000.<br>
```

n

Index

End of PMSE Manual (as of October 12, 2025)