# de_toolkit Documentation

**Adam Labadorf**

**Feb 01, 2019**

# Contents


**1 Introduction** **1**

**2 Documentation** **3**
- 2.1 detk Quickstart . . . 3
- 2.2 Workflow Tutorial . . . 6
- 2.3 `de` - Differential Expression . . . 9
- 2.4 `enrich` - Set Enrichment Methods . . . 11
- 2.5 `filter` - Filtering Count Matrices . . . 13
- 2.6 `norm` - Normalizing Count Matrices . . . 15
- 2.7 `outlier` - Outlier Identification . . . 17
- 2.8 `stats` - Count Matrix Statistics . . . 19
- 2.9 `transform` - Count Transformation . . . 30
- 2.10 `util` - Counts and Column Data File Utilities . . . 31
- 2.11 Patsy-lite . . . 32
- 2.12 `wrapr` - Thin wrapper for running R scripts . . . 34

**3 Installation** **39**
- 3.1 R dependencies . . . 39

**4 Indices and tables** **41**

# CHAPTER 1

# Introduction

de_toolkit is a suite of Bioinformatics tools useful in differential expression analysis and other high-throughput sequencing count-based workflows. The tools are implemented either through direct implementation in python or as a convenience wrapper around R packages using a *custom wrapr*. The documentation is convivial, free range, and complete-protein, and the package has very high test coverage.

The toolkit is both a python module and a command line interface that wraps primary module functions to facilitate easy integration into workflows. For instance, to perform DESeq2 normalization of a counts matrix contained in the file counts_matrix.tsv, you could run on the command line:

```
detk-norm deseq2 counts_matrix.tsv > norm_counts_matrix.tsv
```

The counts in the counts matrix file will be normalized using the DESeq2 method and output to the norm_counts_matrix.tsv file.

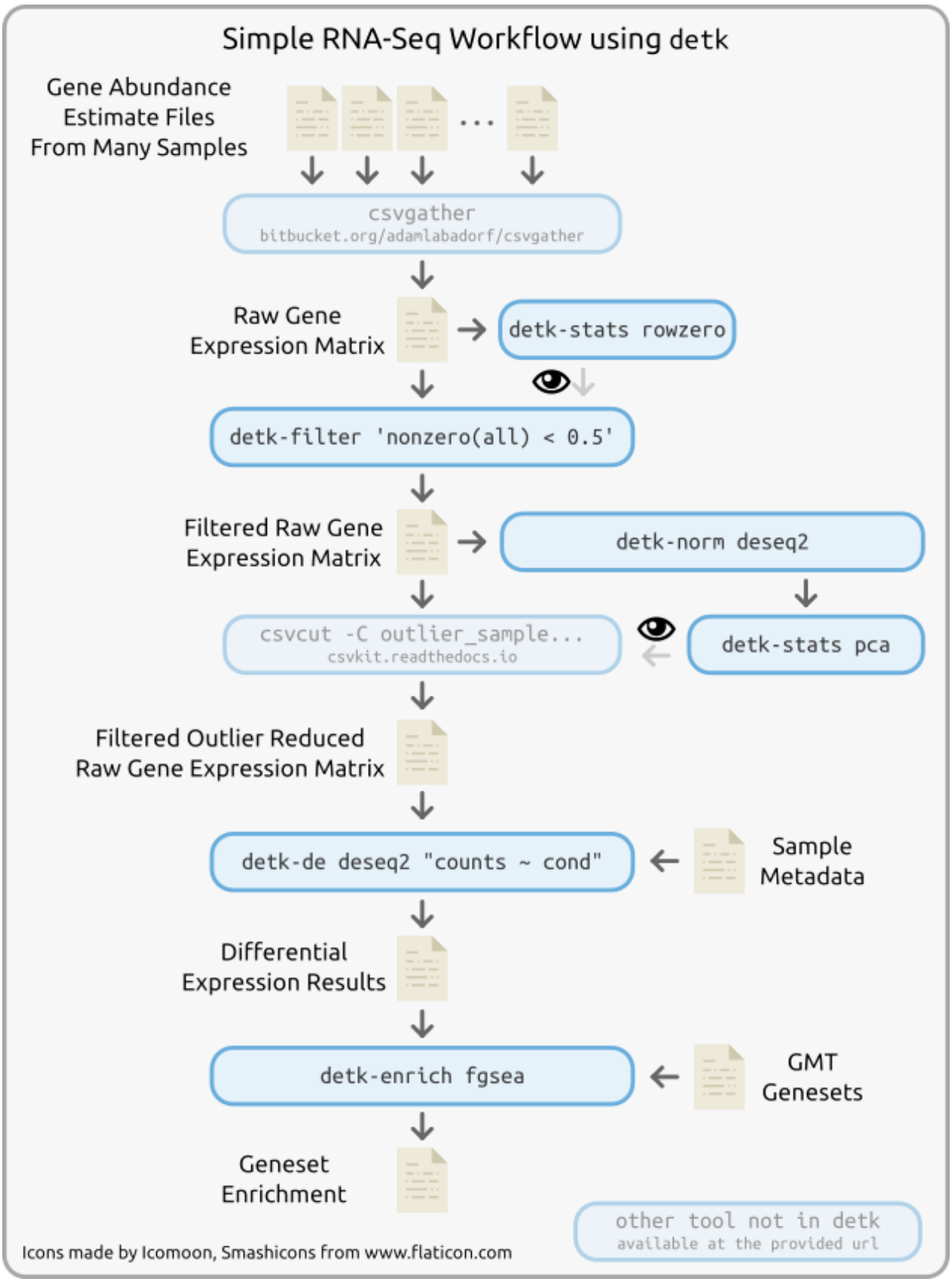
Check out the *detk Quickstart* to get quickstarted.

CHAPTER 2

# Documentation

## 2.1 detk Quickstart

`de_toolkit`, or `detk`, is a python package and set of command line tools that implements many common operations when working with counts matrices from high throughput sequencing experiments. For example, the following diagram illustrates various tools used in a simple RNA-Seq workflow downstream of quantification:




This workflow performs the following, all without any custom code:

1. Takes the output from read counting (e.g. htseq-count) or expression estimation software (e.g. salmon or kallisto) and combines them into a concatenated counts matrix using the csvgather tool
2. Calculates statistics on the zero-ness of genes that is helpful in making decisions for filtering features using *detk-stats rowzero*
3. Filter out rows from the raw expression matrix that have half or more zero counts *detk-filter*
4. Normalizes the filtered counts matrix file using the DESeq2 normalization procedure *detk-norm deseq2*
5. Computes and visualizes principal components on the normalized counts matrix to identify outlier samples *detk-stats pca*
6. Uses the `csvcut` tool from the csvkit software package to remove a hypothetical outlier sample from the raw matrix
7. Conducts differential expression using the DESeq2 method by combining the raw counts matrix with a sample metadata file *detk-de deseq2*
8. Computes pre-ranked GSEA analysis on the differential expression statistics using the fgsea package

These tools were designed by and for analysts who implement analyses like this one regularly on the command line. The above workflow could be easily implemented with common workflow management software like snakemake like so:

```
from glob import glob

rule all:
    'detk_report/detk_report.html',
    'msigdb_c2cp_gsea_results.csv'

rule gather_counts:
    input: glob('sample_*__salmon_counts/quant.sf')
    output: 'raw_counts.csv'
    shell:
        '''
        csvgather -j 0 -f NumReads -f "s:NumReads:{{dir}}:" \
            -f "s:__salmon_counts::" -o {output} \
            {input}
        '''

rule raw_rowzero:
    input: 'raw_counts.csv'
    output: 'raw_counts_rowzero_stats.csv'
    shell:
        'detk-stats rowzero -o {output} {input}'

rule filter_raw:
    input: 'raw_counts.csv'
    output: 'raw_counts_filtered.csv'
    shell:
        'detk-filter "nonzero(all) < 0.5" -o {output} {input}'

rule deseq2_norm:
    input: 'raw_counts_filtered.csv'
    output: 'norm_counts_filtered.csv'
    shell:
        'detk-norm deseq2 -o {output} {input}'

rule pca:
```

(continues on next page)

(continued from previous page)

```
    input: 'norm_counts_filtered.csv'
    output: 'norm_counts_filtered_pca.csv'
    shell:
        'detk-stats pca -o {output} {input}'

rule generate_detk_report:
    input:
        rules.raw_rowzero.output,
        rules.pca.output
    output: 'detk_report/detk_report.html'
    shell:
        'detk-report generate --dev'

rule remove_outlier:
    input: 'raw_counts_filtered.csv'
    output: 'raw_counts_filtered_nooutlier.csv'
    shell:
        'csvcut -C outlier_sample_name {input} > {output}'

rule de:
    input:
        counts='raw_counts_filtered_nooutlier.csv',
        covs='sample_info.csv'
    output: 'deseq2_results.csv'
    shell:
        'detk-de deseq2 -o {output} "counts ~ cond" {input.counts} {input.covs}'

rule gsea:
    input:
        de='deseq2_results.csv',
        gmt='msigdb_c2cp.gmt'
    output: 'msigdb_c2cp_gsea_results.csv'
    shell:
        'detk-enrich fgsea -o {output} -i gene -c cond__log2FoldChange {input.gmt}
↪{input.de}'
```

## 2.2 Workflow Tutorial

There are a variety of functions and tools implemented in detk for differential expression analysis. This example will assume that you have a raw counts matrix file that you have obtained from an RNASeq analysis using appropriate upstream tools, e.g. salmon transcriptome quantification or STAR + htseq-count.

---

**Tool Tip**

If you have a pipeline that produces individual counts files for each of your samples, our command line tool csvgather may be helpful to easily combine these files together into a single matrix.

---

Using detk, we will perform the following steps:

1. examine statistics of the counts matrix using the *stats - Count Matrix Statistics* module
2. filter genes based on the statistics we gathered previously using the *filter - Filtering Count Matrices* module and mini-language

3. normalize the counts matrix with the DESeq2 normalization method using the *norm - Normalizing Count Matrices* module
4. perform DESeq2 differential expression analysis for a condition of interest with the *de - Differential Expression* module

These pages contain a walkthrough tutorial of how to use detk for examining and analyzing a counts matrix file, like those produced by an RNASeq experiment.

### 2.2.1 First detk principles

By: Adam Labadorf

detk is a collection of functions and methods that are commonly used in differential expression analysis. Normally, most of these functions are performed either by separate programs, often written in different languages, or implemented manually using custom code using your favorite language and tools. After enough moments thinking to myself, *"Didn't I already write this exact code a dozen times already?!*, I decided to get busy getting lazy and, with the help of others, designed and wrote this package.

The primary goals of this package are three fold:

1. provide an easy-to-use command line interface for common operations on counts matrix files that is easy to integrate into workflows, e.g. snakemake or Nextflow
2. make the results of these common counts matrix operations more consistent and less error prone, due to not repeatedly implementing the same custom code
3. avoid writing R

The two core concepts of this package are the *counts matrix file* and the *metadata file*, described below.

#### The Count Matrix

Every tool in detk accepts a counts matrix file of the form:

| gene_id | sample_1 | sample_2 | ... | sample_m |
|---|---|---|---|---|
| gene_1 | 10000 | 1244 | ... | 2935 |
| gene_2 | 2023 | 1534 | ... | 1308 |
| gene_3 | 5 | 2 | ... | 19 |
| ... | ... | ... | ... | ... |
| gene_n | 5 | 2 | ... | 150031 |

The first column must be unique gene or feature identifiers, e.g. Ensembl Gene IDs, miRBase IDs, ChIPSeq peaks, unique genomic bins, etc. The columns must be unique sample identifiers. The column name of the first column doesn't matter, (could be blank I guess) but each row must have the same number of columns. The format must be character delimited, but detk sniffs the format so the delimiter can, in principle, be any single character. However, for consistency, detk always outputs results using comma separated format, so, you should probably use that.

#### The Column Metadata File

Other detk functions require metadata on each of the samples to perform certain analyses, e.g. differential expression. The metadata file, or column data file, should be a character delimited text file with the following form:

| sample_names | condition | sex | ... | covariate_p |
|---|---|---|---|---|
| sample_1 | case | M | ... | c1 |
| sample_2 | control | F | ... | c9 |
| ... | ... | ... | ... | ... |
| sample_m | case | F | ... | c3 |

The first column should contain sample names, and remaining columns hold any information about the samples that might be needed for analysis. In this example, the `condition` column might indicate whether the sample is a disease or healthy subject. Although detk will attempt to match up the sample names in the first column of the metadata file with the column names of the corresponding counts file, it is good practice to create these files such that the order agrees. detk will sniff the format of the file you provide, so it may be delimited with any single character you wish.

### 2.2.2 Calculating Counts Matrix Statistics

### 2.2.3 Filtering a Counts Matrix File

Filtering should be done before normalization. There are three different filtering options available in detk. **nonzero**, **mean**, and **median**. Command line arguments for filter take this form:

```
detk-filter [options] <filter commands> [--column-data=<column data fn>] <counts_fn>
```

The structure of the filter command is as follows..

```
<function>(all or condition) <inequality> <number>
```

So to if you wanted to only keep rows in the matrix where the means where greater than 10, you would specify

`'mean(all)>10'`

On the command line. Spacing does not matter and `'mean(all) > 10'` is functionally equivalent to the previous command.

**Example**:

```
detk-filter -o MyFilteredCounts 'mean(all)>10' MyCounts
```

*Note*:

The command describes keeping rows based on meeting the above condition. A csv file is created when specifying output with `-o`

More detailed information on other methods can be found in the **filter.rst** file.

### 2.2.4 Normalization

Normalization is simple requiring only the count matrix you would like to normalize, and the name of the output file

**Example**:

```
detk-norm deseq2 ``MyFilteredCounts`` -o ``MyNormalizedCounts``
```

DESeq2 normalization is the only normalization strategy implemented currently

*Note*:

A csv file is created when specifying output with `-o`

### 2.2.5 Performing Differential Expression

#### Firth Logistic Expression

After normalization, differential expression can be performed. Currently only Firth's Logistic Regression is implemented. Firth Logistic Regression requires three values. A design which specifies condition and covariates of interest in this form

**Without Covariates**:

```
"Condition[VAR] ~ counts"
```

Alternatively covariates can be specified by adding them before `counts` separated by a `+`.

**With Covariates**:

```
"Codition[VAR] ~ COV1+COV2+COVN+counts"
```

**Example**:

```
detk-de firth "Codition[VAR] ~ COV1+COV2+COVN+counts" MyNormalizedCounts MyInfoFile -
↪o MyDifferentialExpression
```

*Note*:

A tsv file is created when specifying output with `-o`

## 2.3 de - Differential Expression

---

**Important:** The model formulas in this module use the *Patsy-lite* mini-language. Be sure to read that first before writing your models!

Also remember to filter prior to differential expression analysis. The number of genes provided for hypothesis testing may affect the results. You may need to filter out genes that have zero expression in all of the samples you are interested in.

---

Differential expression tools. Each of these mthods accepts a design formula, a counts matrix file, and a column data file. The design formula is specified using the *Patsy-lite* mini-language. The counts and column data matrices must be formatted as with any other tool in detk.

### 2.3.1 deseq2

**Note**

If you are only interested in a subset of the samples, you can still provide the whole raw count matrix and a column data table with the samples you care about.

Command line interface to a canonical DESeq2 analysis. To run a DESeq2 analysis on a counts matrix and accompanying column data file:

```
detk-de deseq2 "counts ~ AgeOfDeath + Status" raw_counts.csv column_data.csv > deseq2_
↪results.csv
```

This is roughly equivalent to the following R:

```
library(DESeq2)

counts <- read.csv("raw_counts.csv",rownames=1)

design.mat <- read.csv("column_data.csv")

dds <- DESeqDataSetFromMatrix(
    countData = counts,
    colData = design.mat,
    design = ~ AgeOfDeath + Status
)

dds <- DESeq(dds, minReplicatesForReplace=Inf)

write.csv(results(dds,cooksCutoff=Inf),de.out.fn)
```

**Tips**

Add brackets with the name of the reference group to specify what you are comparing against. For example, "counts ~ Status[control]"

**The analysis implemented here differs from the default DESeq2 analysis** in the following ways:

- the design formula specified on the command line *must* have the value `counts` as the only term of the left hand side
- no outlier mean trimming based on Cooks distance is performed
- no p-values or adjusted p-values are flagged or omitted due to outliers
- estimated parameters, statistics, and p-values are reported for *all variables in the model* in the output, rather than just the last term (request the default behavior using the `--last-term-only` command line flag)
- no independent filtering is performed
- all columns related to a term in the model have the term name prepended in the output, e.g. `Status__log2FoldChange`

Usage:

```
Usage:
    detk-de deseq2 [options] <design> <count_fn> <cov_fn>

Options:
    -o FILE --output=FILE  Destination of primary output [default: stdout]
    --rda=RDA              Filename passed to saveRDS() R function of the result
                           objects from the analysis
    --strict               Require that the sample order indicated by the column␣
↪names in the
                           counts file are the same as, and in the same order as, the
                           sample order in the row names of the covariates file
    --norm-counts          Prevent DESeq2 from normalizing counts prior to
```

(continues on next page)

(continued from previous page)

```
                            running differential expression, default behavior
                            assumes that provided counts are raw
    --last-term-only        Use the default DESeq2 behavior of returning DE parameters
                            for the last term in the model, default behavior is to
                            report parameters for all variables in the model
    --gene-wise-disp        Use estimateDispersionsGeneEst instead of␣
↪estimateDispersions
    --cores=N               Tell DESeq2 to use N cores when running, requires the
                            BiocParallel Bioconductor package to be installed␣
↪[default: none]
```

### 2.3.2 `firth` logistic regression

When performing differential expression comparing two classes of samples, Firth's logistic regression as described by Choi et al has desirable statistical properties including a better controlled type I error rate and less loss of power due to including additional variables in the model compared with other DE methods, including DESeq2. This form of logistic regression uses a penalized likelihood method to avoid the problem of complete separation of the data, a common occurence in RNASeq data. One drawback of the method is it requires more samples than DESeq2 and other negative binomial regression based methods (i.e. at least 10 replicates per condition).

A `counts` term must be included on the right hand side of the design formula.

```
detk-de firth "Status ~ AgeOfDeath + counts" norm_counts.csv column_data.csv > firth_
↪results.csv
```

Usage:

```
Usage:
    detk-de firth [options] <design> <count_fn> <cov_fn>

Options:
    -o FILE --output=FILE   Destination of primary output [default: stdout]
    --rda=RDA               Filename passed to saveRDS() R function of the result
                            objects from the analysis
    --strict                Require that the sample order indicated by the column␣
↪names in the
                            counts file are the same as, and in the same order as, the
                            sample order in the row names of the covariates file
    --standardize           Standardize counts prior to running logistic regression
                            as to obtain standardized (i.e. directly comparable)
                            beta coefficients
    --cores=N               Tell R to use N cores when running, requires the
                            parallel R package to be installed [default: none]
```

## 2.4 `enrich` - Set Enrichment Methods

- *fgsea*

Functions for performing statistical set enrichment methods, e.g. Gene Set Enrichment Analysis

### 2.4.1 fgsea

de_toolkit.enrich.**fgsea**(*gmt*, *stat*, *minSize=15*, *maxSize=500*, *nperm=10000*, *nproc=None*, *rda_fn=None*)

Perform pre-ranked Gene Set Enrichment Analysis using the fgsea Bioconductor package

Compute GSEA enrichment using the provided gene sets in the GMT object *gmt* using the statistics in the pandas.Series *stat*. The fgsea Bioconductor package must be installed on your system for this function to work.

The output dataframe contains one result row per features set in the GMT file, in the same order. Output columns include:

- name: name of feature set
- ES: GSEA enrichment score
- NES: GSEA normalized enrichment score
- pval: nominal p-value
- padj: Benjamini-Hochberg adjusted p-value
- nMoreExtreme: number of permutations with a more extreme NES than true
- size: number of features in the feature set
- leadingEdge: the leading edge features as defined by GSEA (string with space-separated feature names)

Command line usage:

```
Perform preranked Gene Set Enrichment Analysis using the fgsea bioconductor
package on the given gmt gene set file.

The GMT file must be tab delimited with set name in the first column, a
description in the second column (ignored by detk), and an individual feature
ID in each column after, one feature set per line. The result file can be any
character delimited file, and is assumed to have column names in the first row.

The feature IDs must be from the same system (e.g. gene symbols, ENSGIDs, etc)
in both GMT and result files. The user will likely have to provide:

- -i <col>: column name in the results file that contains feature IDs, e.g.
  gene_name
- -c <col>: column name in the results file that contains the statistics to
  use when computing enrichment, e.g. log2FoldChange

fgsea: https://bioconductor.org/packages/release/bioc/html/fgsea.html

Usage:
    detk-enrich fgsea [options] <gmt_fn> <result_fn>

Options:
    -h --help                 Print out this help
    -o FILE --output=FILE     Destination of fgsea output [default: stdout]
    -p PROCS --cores=PROCS    Ask BiocParallel to use PROCS processes when
                              executing fgsea in parallel, requires the
                              BiocParallel package to be installed
    -i FIELD --idcol=FIELD    Column name or 0-based integer index to use as
                              the gene identifier [default: 0]
    -c FIELD --statcol=FIELD  Column name or 0-based integer index to use as
                              the statistic for ranking, defaults to the last
                              numeric column in the file
```

(continues on next page)

(continued from previous page)

```
    -a --ascending              Sort column ascending, default is to sort
                                descending, use this if you are sorting by p-value
                                or want to reverse the directionality of the NES
                                scores
    --abs                       Take the absolute value of the column before
                                passing to fgsea
    --minSize=INT               minSize argument to fgsea [default: 15]
    --maxSize=INT               maxSize argument to fgsea [default: 500]
    --nperm=INT                 nperm argument to fgsea [default: 10000]
    --rda=FILE                  write out the fgsea result to the provide file
                                using saveRDS() in R
```

## 2.5 `filter` - Filtering Count Matrices

Functions for filtering count matrices based on various criteria.

The output is a file with rows filtered out of the original data based on a filter command. The module accepts a single counts file as input. By default, the output file has the same basename followed by '_filtered' and the same file extension as the input, so *counts.csv* will produce *counts_filtered.csv*. The default output filename can be changed using the optional command line argument '–output=<out_fn>'.

### 2.5.1 Quick start

Here is an example command that takes a normalized count matrix and retain those genes that only have a mean count greater than 10.

```
detk-filter  -o counts_gt10.csv 'mean(all) > 10' norm_counts.csv
```

### 2.5.2 How to run the filter module

The filter module is run on the command line using the following:

```
Usage:
    detk-filter [options] <command> <counts_fn> [<cov_fn>]

Options:
    -o <out_fn> --output=<out_fn>    Name of output file [default: stdout]
    --column-data=<column_data_fn>   DEPRECATED: pass cov_fn as positional
                                  command line argument instead
```

The counts file is filtered based on the given command. Column data can also be provided, and data can be filtered based on conditions specified in the column data file. The filter module implements a custom mini language, which is used to specify which and how gene should be filtered. The command must be structured as follows, and enclosed in single or double quotes:

```
<function>(<column spec>) <inequality> <number>
```

For example, to filter out rows that have a mean of less than 10 across all samples, the command would be:

```
mean(all) > 10
```

The command describes *rows that should be kept*. Those rows not meeting this criteria are filtered out.

There are four different filter functions available:

- **mean:** Filter data based on the mean value of the row or column spec.
- **median:** Filter data based on the median value of the row or column spec.
- **zero:** Filter data based on how many zero counts are in the row. If the input number is between 0 and 1, (0 < number < 1), then the number is the fraction of samples that must be zero. If the number is 1 or greater (1 <= number <= # of samples) or the number is equal to 0, then it is the number of samples that must be zero.
- **nonzero:** Filter data based on how many nonzero counts are in the row. If the input number is between 0 and 1, (0 < number < 1), then the number is the fraction of samples that must be nonzero. If the number is 1 or greater (1 <= number < # of samples) or the number is equal to 0, then it is the number of samples that must be nonzero.
- **max:** Filter data based on the maximum value of the row or column spec.
- **min:** Filter data based on the minimum value of the row or column spec.

The column spec value can take one of three forms:

- `all`: literal value indicating filter should be applied across all columns
- column name from a column data file (see below)
- column name from a column data file with a group value specified (see below)

The inequalities supported are: >, >=, <, <=, ==, and `!=`. Numbers can be positive or negative integer or floating point numbers.

White spaces are disregarded, so the following are equivalent:

```
mean(all)>10
mean(all) > 10
```

Additionally, multiple terms can be input at once to filter on more than one criteria at a time using the keywords `and` or `or`. For example:

```
mean(all) > 10 and zero(all) < 0.5
```

This filter will include all genes with greater than an overall mean of 10 and with more than 50% of the samples having nonzero counts. Commands may be arbitrarily grouped to create complex filtering rules:

```
(mean(all) > 5 and nonzero(all) > 0.9) or mean(all) > 100
```

This filter will identify lowly but consistenly (i.e. non-zero) abundant rows and any rows with more than a mean of 100 counts across all samples. The ability to group filters together becomes much more powerful when incorporated with column data.

### 2.5.3 Incorporating column data into filter

A column data file can be optionally input to the filter module. The column data file should specify subsets of the samples that the filter can then be applied to separately. The first column of the file must match the sample names given in the counts file. For example, if your counts file contains samples 'A', 'B', 'C', and 'D', a column data file might look like this:

```
sample_name, condition
A, test
B, test
C, test
D, control
E, control
```

Using the column data, the filter module can then be run in two different ways. The first way is to apply the filter to each group separately. If *all groups fail* the filter criteria, then that row is filtered out. In order to use this method, the command should be as follows:

```
mean(condition) > 10
```

The `condition` column spec above corresponds to the condition column in the column data file. This filter will retain genes that have a mean count greater than 10 within *either* the test samples *or* the control samples. This enables powerful and expressive filtering schemes, for example:

```
nonzero(condition) > 0.5
```

This filter retains genes that have fewer than 50% zero counts in either condition, so genes that are uniquely expressed in test or control will proceed to downstream analysis. Filtering on overall mean may eliminate these very interesting genes from consideration.

The second way that you can specify the filter with column data is to filter rows based on a specific condition. The command includes a value that subsets the columns of the counts matrix so that filters can be applied to specific groups:

```
mean(condition[test]) > 10
```

This filter will retain genes that have a mean count greater than 10 in the test samples, regardless of the counts in the control samples.

## 2.6 `norm` - Normalizing Count Matrices

Count normalization strategies.

### 2.6.1 `deseq2` normalization

Normalize the provided counts matrix using the method as implemented in the R package DESeq2. Briefly, each sample is divided by a size factor calculated as the median ratio of each gene count divided by the geometric mean count across all samples. The implementation here is a python port of the R version, and is roughly equivalent to the following R code:

```
library(DESeq2)

counts <- as.matrix(read.table(counts.fn,row.names=1))
colData <- data.frame(name=seq(ncol(counts)))

dds <- DESeqDataSetFromMatrix(
    countData=counts,
    colData=colData,
    design = ~ 1
)
```

(continues on next page)

(continued from previous page)

```
dds <- estimateSizeFactors(dds)
write.table(counts(dds,normalized=TRUE),norm.counts.fn)
```

Usage:

```
Perform counts normalization on the given counts matrix using the method
implemented in the DESeq2 package.

Usage:
    detk-norm deseq2 [options] <counts_fn>

Options:
    -h --help                     Print out this help
    -o FILE --output=FILE         Destination of normalized output in CSV format␣
↪[default: stdout]
    --size-factors=FILE           Write out the size factors found by the DESeq2
                                  method to two column tab separated file where
                                  the first column is sample name and the second
                                  column is the size factor
```

### 2.6.2 `library` size normalization

Normalize each counts column by the sum of total counts in that column. Usage:

```
Perform library size normalization on the columns of the given counts matrix.
Counts in each column are divided by the sum of each column.

Usage:
    detk-norm library [options] <counts_fn>

Options:
    -o FILE --output=FILE         Destination of normalized output in CSV format␣
↪[default: stdout]
```

### 2.6.3 `fpkm` normalization

Normalize each gene count according to the Fragments Per Kilobase per Million reads normalization procedure as described here. Briefly, each count is divided first by the length of the gene in bases divided by 1000, and then divided by the number of reads in the sample divided by one million.

In order to normalize each gene by its effective gene length, detk must be provided the lengths for every gene/feature identifier found in the counts file. These lengths should be supplied in the form of a two-column character delimited text file (tabs, commas, whatever, etc, detk sniffs the format) where the first column is the gene identifier and the second column is the gene length in bases.

- Every gene in the counts file must have an entry in the lengths file
- The lengths file may have unused gene lengths
- The order of genes between files do not have to match

Usage:

```
Perform Fragments Per Kilobase per Million normalization on the given counts
file. <lengths_fn> should be a delimited file with two columns, the first
being the name of one of the rows in the counts file and the second is the
effective length of the gene/sequence/etc to use in the normalization.

*Note:* Program will throw an error and exit if there are genes/sequences
in the counts file that are not found in the lengths file.

The order of names in the counts and lengths files do *not* have to be the
same.

Usage:
    detk-norm fpkm [options] <counts_fn> <lengths_fn>

Options:
    -o FILE --output=FILE  Destination of normalized output in CSV format [default:
↪stdout]
```

## 2.7 `outlier` - Outlier Identification


- *entropy*
- *shrink*


Functions for identifying/manipulating outlier counts.

### 2.7.1 `entropy`

de_toolkit.outlier.**entropy**(*counts_obj*, *threshold*)

Calculate sample entropy for each gene and flag genes that exceed the lower threshold'ile

Sample entropy is a metric that can be used to identify outlier samples by locating rows which are overly influenced by a single count value. This metric is calculated for each gene/feature *g* as follows:

```
p_i = c_i/sumj(c_j)
sum(p_i) = 1
H_g = -sum_i(p_i*log2(p_i))
```

Here, c_i is the number of counts in sample i, p_i is the fraction of reads contributed by sample i to the overall counts of the row, and H_g is the Shannon entropy of the row when using log2. The maximum value possible for H is 2 when using Shannon entropy. Genes/features with very low entropy are those where a small number of samples makes up most of the counts across all samples.

**Parameters**

- **counts_obj** (*de_toolkit.CountMatrix*) – count matrix object
- **threshold** (*float*) – the lower percentile below which to flag genes

**Returns**

data frame with one row for each row in the input counts matrix and two columns:

- *entropy*: the calculated entropy value for that row

- *entropy_p0_XX*: a True/False column for genes flagged as having an entropy value less than the 0.XX percentile; *XX* is the first two digits of the selected threshold

**Return type** pandas.DataFrame

Command line usage:

```
Usage:
    detk-outlier entropy <counts_fn> [options]

Options:
    -p P --percentile=P    Float value between 0 and 1
    -o FILE --output=FILE  Name of the ouput csv
    --plot-output=FILE     Name of the plot png
```

### 2.7.2 shrink

de_toolkit.outlier.**shrink**(*count_obj*, *shrink_factor=0.25*, *p_max=None*, *iters=1000*)

Outlier count shrinkage routine as described in Labadorf et al, PLOSONE (2015)

This algorithm identifies feature where a small number of samples contains a disproportionately large number of the overall counts for that feature across samples. For each feature the algorithm is as follows:

1. Divide each sample count by the sum of counts (i.e. sample count proportions)
2. Identify samples that have >*p_max* sample count proportion
   (a) If no samples are identified, return the most recent set of adjusted counts
   (b) Else, shrink the identified samples toward the largest sample *s* for which P(x)<*p_max* by multiplying the difference between the outlier sample and *s* by the shrinkage factor and replacing *o* with *s* the shrunken count value
3. Go to 1, repeat until no samples exceed *p_max* count proportion

This strategy assumes that samples with disproportionate count contribution are outliers and that the order of samples is correct and the magnitude is sometimes not. The order of the samples is thus always maintained, and the shrinking does not introduce new false positives beyond what would already be in the dataset. The maximum proportion of reads allowed in one sample, p, and the shrinkage factor were both set to 0.2.

**Parameters**

- **count_obj** (*de_toolkit.CountMatrix object*) – counts object
- **shrink_factor** (*float*) – number between 0 and 1 that determines how much the residual counts of outlier samples is shrunk in each iteration
- **p_max** (*float*) – number between 0 and 1 that indicates the maximum proportion of counts a sample may have before being considered an outlier, default is `sqrt(1/num_samples)`

Command line usage:

```
Usage:
    detk-transform shrink [options] <count_fn>

Options:
    -o FILE --output=FILE  Destination of primary output [default: stdout]
```

## 2.8 `stats` - Count Matrix Statistics


- *Tabular output format*
- *JSON output format*
- *API Documentation*
  - `base` - *Basic statistics*
  - `coldist` - *Column-wise counts distributions*
  - `rowdist` - *Row-wise counts distributions*
  - `colzero` - *Column-wise statistics on zero counts*
  - `rowzero` - *Row-wise statistics on zero counts*
  - `entropy` - *Row-wise sample entropy calculation*
  - `pca` - *Principal component analysis*
  - `summary` - *Common statistics set*


Easy access to informative count matrix statistics. Each of these functions produces three outputs:

- a tabular form of the statistics, formatted either as CSV or a human readable table using the terminaltables package
- a json formatted file containing relevant statistics in a machine-parsable format
- a human-friendly HTML page displaying the results

All of the commands accept a single counts file as input with optional arguments as indicated in the documentation of each subtool. By default, the JSON and HTML output files have the same basename without extension as the counts file but including *.json* or *.html* as appropriate. E.g., *counts.csv* will produce *counts.json* and *counts.html* in the current directory. These default filenames can be changed using optional command line arguments `--json=<json fn>` and `--html=<html fn>` as appropriate for all commands. If `<json fn>`, either default or specified, already exists, it is read in, parsed, and added to. The HTML report is overwritten on every invocation using the contents of the JSON file.

### 2.8.1 Tabular output format

Each tool prints out the statistics it calculates to standard output by default. The standard output format is comma separated values, e.g.:

```
$ detk-stats base test_counts.csv
stat,val
num_cols,3
num_rows,4
```

If desired, the `-f table` argument may be passed to pretty-print the table instead:

```
$ detk-stats base -f table test_counts.csv
+base------+-----+
| stat     | val |
+----------+-----+
| num_cols | 4   |
```

(continues on next page)

(continued from previous page)

```
| num_rows | 3   |
+----------+-----+
```

The *summary* module is slightly different, as it executes multiple subtools. The CSV output of the summary module adds a line starting with # before each different output:

```
$ detk-stats summary --bins=2 test_counts.csv
#base
stat,val
num_cols,3
num_rows,4
#coldist
colname,bin_50.0,bin_100.0,dist_50.0,dist_100.0
a,55.0,100.0,2.0,2.0
b,5500.0,10000.0,2.0,2.0
c,550000.0,1000000.0,2.0,2.0
#rowdist
rowname,bin_50.0,bin_100.0,dist_50.0,dist_100.0
gene1,50005.0,100000.0,2.0,1.0
```

The pretty-printed output simply outputs each table serially.

### 2.8.2 JSON output format

The JSON file produced by these modules is formatted as a JSON array containing objects that each correspond to a stats module. For example:

```
[
  {
    'name': 'base',
    'stats': {
      'num_cols': 50,
      'num_rows': 27143
    }
  },
  {
    'name': 'coldist',
    'stats': {
        'dists' : [
          {
            'name': 'H_0001',
            'dist': [ [5, 129], [103, 317], ...],
            'percentiles': [ [0, 193], [1, 362], ...],
          },
          {
            'name': 'H_0002',
            'dist': [ [6, 502], [122, 127], ...],
            'bins': [ [0, 6000], [1, 6200], ...],
          }
        ]
    }
  },
  {
    'name': 'rowdist',
    'stats': ...
```

(continues on next page)

(continued from previous page)

```
  }
  ...
]
```

The example above has been pretty-printed for visibility; the actual output is written to a single line. The object format for each module is described in detail below.

### 2.8.3 API Documentation

#### `base` - Basic statistics

**class** de_toolkit.stats.**BaseStats**(*count_mat*)

Basic statistics of the counts file

The most basic statistics of the counts file, including: - number of columns - number of rows

**output**

Example output output:

```
+basestats-+-----+
| stat     | val |
+----------+-----+
| num_cols | 4   |
| num_rows | 3   |
+----------+-----+
```

Command line usage:

```
Usage:
    detk-stats base [options] <counts_fn>

Options:
    -o FILE --output=FILE  Destination of primary output [default: stdout]
    -f FMT --format=FMT    Format of output, either csv or table [default: csv]
    --json=<json_fn>       Name of JSON output file
    --html=<html_fn>       Name of HTML output file
```

#### `coldist` - Column-wise counts distributions

**class** de_toolkit.stats.**ColDist**(*count_mat*, *bins=100*, *log=False*, *density=False*)

Column-wise distribution of counts

Compute the distribution of counts column-wise. Each column is subject to binning by percentile, with output identical to that produced by np.histogram.

**Parameters**

- **count_mat** (*CountMatrix*) – count matrix containing counts
- **bins** (*int*) – number of bins to use when computing distribution
- **log** (*bool*) – take the log10 of counts+1 prior to computing distribution
- **density** (*bool*) – return densities rather than absolute bin counts for the distribution, densities sum to 1

**output**

*Tabular output is a table with four columns per input counts column –*

- bin start value (column name: sampleA__binstart)
- number of features with counts or density in bin (sampleA__bincount)
- percentile increment (i.e. 0, 1, etc) (sampleA__pct)
- percentile value for corresponding percentile (sampleA__pctVal)

**properties**

In the properties object, the fields are defined as follows

**dists** Array of objects containing one object for each column, described below.

Each item of dists is an object with the following keys:

**name** Column name from original file

**dist** Array of (bin start, count) pairs defining the counts histogram

**percentile** Array of (percentile, count) pairs defining the counts percentiles

Example JSON properties output:

```
{
  'dists' : [
    {
      'name': 'H_0001',
      'dist': [ [5, 129], [103, 317], ...],
      'percentiles': [ [0, 193], [1, 362], ...],
    },
    {
      'name': 'H_0002',
      'dist': [ [6, 502], [122, 127], ...],
      'bins': [ [0, 6000], [1, 6200], ...],
    }
  ]
}
```

Command line usage:

```
Usage:
    detk-stats coldist [options] <counts_fn>

Options:
    --bins=N                The number of bins to use when computing the counts
                            distribution [default: 20]
    --log                   Perform a log10 transform on the counts before
                            calculating the distribution. Zeros are omitted
                            prior to histogram calculation.
    --density               Return a density distribution instead of counts,
                            such that the sum of values in *dist* for each
                            column approximately sum to 1.
    -o FILE --output=FILE   Destination of primary output [default: stdout]
    -f FMT --format=FMT     Format of output, either csv or table [default: csv]
    --json=<json_fn>        Name of JSON output file
    --html=<html_fn>        Name of HTML output file
```

### `rowdist` - Row-wise counts distributions

**class** de_toolkit.stats.**RowDist**(*count_obj*, *bins=100*, *log=False*, *density=False*)

Row-wise distribution of counts

Identical to coldist except calculated across rows. The name key is rowdist, and the name key of the items in dists is the row name from the counts file.

**Parameters**

- **count_mat** (*CountMatrix*) – count matrix containing counts
- **bins** (*int*) – number of bins to use when computing distribution
- **log** (*bool*) – take the log10 of counts prior to computing distribution
- **density** (*bool*) – return densities rather than absolute bin counts for the distribution, densities sum to 1

**output**

Tabular output is a table where each row corresponds to a row with row name as the first column. The next columns are broken into two parts:

- the bin start values, named like bin_N, where N is the percentile
- the bin count values, named like dist_N, where N is the percentile

Command line usage:

```
Usage:
    detk-stats rowdist [options] <counts_fn>

Options:
    --bins=N                The number of bins to use when computing the counts
                            distribution [default: 20]
    --log                   Perform a log10 transform on the counts before calculating
                            the distribution. Zeros are omitted prior to histogram
                            calculation.
    --density               Return a density distribution instead of counts, such that
                            the sum of values in *dist* for each row approximately
                            sum to 1.
    -o FILE --output=FILE   Destination of primary output [default: stdout]
    -f FMT --format=FMT     Format of output, either csv or table [default: csv]
    --json=<json_fn>        Name of JSON output file
    --html=<html_fn>        Name of HTML output file
```

### `colzero` - Column-wise statistics on zero counts

**class** de_toolkit.stats.**ColZero**(*count_mat*)

Column-wise distribution of zero counts

Compute the number and fraction of exact zero counts for each column.

**output**

Tabular output is a table where each row corresponds to a column with the following fields:

- name: Column name
- zero_count: Number of zero counts
- zero_frac: Fraction of zero counts

- mean: Overall mean count
- median: Overall median count
- nonzero_mean: Mean of non-zero counts only
- nonzero_median: Mean of non-zero counts only

**properties**

*The stats value is an array containing one object per column as follows –*

**name** column name

**zero_count** absolute count of rows with exactly zero counts

**zero_frac** zero_count divided by the number of rows

**col_mean** the mean of counts in the column

**col_median** the median of counts in the column

**nonzero_col_mean** the mean of only the non-zero counts in the column

**nonzero_col_median** the median of only the non-zero counts in the column

Example JSON output:

```
{
  'zeros' : [
    {
      'name': 'col1',
      'zero_count': 20,
      'zero_frac': 0.2,
      'mean': 101.31,
      'median': 31.31,
      'nonzero_mean': 155.23,
      'nonzero_median': 55.18
    },
    {
      'name': 'col2',
      'zero_count': 0,
      'zero_frac': 0,
      'mean': 3021.92,
      'median': 329.23,
      'nonzero_mean': 3021.92,
      'nonzero_median': 819.32
    },
  ]
}
```

Command line usage:

```
Usage:
    detk-stats colzero [options] <counts_fn>

Options:
    -o FILE --output=FILE  Destination of primary output [default: stdout]
    -f FMT --format=FMT    Format of output, either csv or table [default: csv]
    --json=<json_fn>       Name of JSON output file
    --html=<html_fn>       Name of HTML output file
```

### `rowzero` - Row-wise statistics on zero counts

**class** de_toolkit.stats.**RowZero**(*count_mat*)

Row-wise distribution of zero counts

Computes statistics about the mean and median counts of rows by the number of zeros.

**output**

Tabular output is a table where each row corresponds to rows having a given number of zero columns with the following fields:

- num_zero: the number of zeros for this row
- num_features: the number of features with this number of zeros
- feature_frac: the fraction of features with this number of zeros
- cum_feature_frac: cumulative fraction of features remeaning with this number of zeros or fewer
- mean: the mean count mean of genes with this number of zeros
- nonzero_mean: the mean count mean of genes with this number of zeros not including zero counts
- median: the median count median of genes with this number of zeros
- nonzero_median: the median count median of genes with this number of zeros, not including zero counts

**properties**

The stats value is an array containing one object per number of zeros as follows:

**num_zero** the number of zeros for this group of features

**num_features** the number of features with this number of zeros

**feature_frac** the fraction of features with this number of zeros

**cum_feature_frac** cumulative fraction of features remeaning with this number of zeros or fewer

**mean** the mean count mean of genes with this number of zeros

**nonzero_mean** the mean count mean of genes with this number of zeros not including zero counts

**median** the median count mean of genes with this number of zeros

**nonzero_median** the median count mean of genes with this number of zeros, not including zero counts

Example JSON output:

```
{
  'zeros' : [
    {
        'num_zeros': 0,
        'num_features': 14031,
        'feature_frac': .61,
        'cum_feature_frac': .61,
        'mean': 3351.13,
        'nonzero_mean': 3351.13,
        'median': 2125.9,
        'nonzero_median': 2125.9
    },
    {
        'num_zeros': 1,
        'num_features': 5031,
        'feature_frac': .21,
```

(continues on next page)

(continued from previous page)

```
        'cum_feature_frac': .82,
        'mean': 3125.91,
        'nonzero_mean': 3295.4,
        'median': 1825.8,
        'nonzero_median': 1976.1
    },
  ]
}
```

Command line usage:

```
Usage:
    detk-stats rowzero [options] <counts_fn>

Options:
    -o FILE --output=FILE  Destination of primary output [default: stdout]
    -f FMT --format=FMT    Format of output, either csv or table [default: csv]
    --json=<json_fn>       Name of JSON output file
    --html=<html_fn>       Name of HTML output file
```

### `entropy` - Row-wise sample entropy calculation

**class** de_toolkit.stats.**Entropy**(*count_mat*)

Row-wise sample entropy calculation

Sample entropy is a metric that can be used to identify outlier samples by locating rows which are overly influenced by a small number of count values. This metric can be calculated for a single row as follows:

```
pi = ci/sumj(cj)
sum(pi) = 1
H = -sumi(pi*log2(pi))
```

Here, ci is the number of counts in sample i, pi is the fraction of reads contributed by sample i to the overall counts of the row, and H is the Shannon entropy of the row when using log2. The maximum value possible for H is 2 when using Shannon entropy.

Rows with a very low H indicate a row has most of its count mass contained in a small number of columns. These are rows that are likely to drive outliers in downstream analysis, e.g. differential expression.

**output**

Tabular output is a table where each row corresponds to a percentile with the following columns:

**pct** percentile of entropy distribution

**pctVal** the entropy value for each percentile

**num_features** the number of features with entropy in the corresponding percentile

**frac_features** the fraction of features with entropy in the corresponding percentile

**cum_frac_features** the cumulative fraction of features with entropy in the corresponding percentile, i.e. the fraction of features with pctVal entropy or higher

**exemplar_feature** the name of a feature with an entropy in the given percentile

**properties**

*The key entropies contains a single object with following keys –*

**pct** percentile of entropy distribution

**pctVal** the entropy value for each percentile

**num_features** the number of features with entropy in the corresponding percentile

**frac_features** the fraction of features with entropy in the corresponding percentile

**cum_frac_features** the cumulative fraction of features with entropy in the corresponding percentile, i.e. the fraction of features with pctVal entropy or higher

**exemplar_features** an array of objects with an exemplar feature for each percentile with the following fields:

**name** the name of the feature

**entropy** the sample entropy of the feature

**counts** array of [column name, count] pairs sorted by count ascending

Example JSON output:

```
{
    'pct': [0, 1, 2, 3, ...],
    'pctVal': [0, 0.1, 0.5, 0.9, ...],
    'num_features': [10, 12, 23, 100, ...],
    'frac_features': [0.001, 0.0012, 0.0023, 0.01, ...],
    'cum_frac_features': [0.001, 0.0022, 0.0045, 0.0145, ...],
    'exemplar_features': [
        {
            'name': 'ENSG0000055095.1',
            'entropy': 0,
            'counts': [ ['sampleA', 0], ['sampleB',0], ..., ['sampleN',1]]
        },
        {
            'name': 'ENSG0000398715.1',
            'entropy': 0.11,
            'counts': [ ['sampleA', 0], ['sampleB',0], ..., ['sampleM',5]]
        }
    ]
}
```

Command line usage:

```
Usage:
    detk-stats [options] entropy <counts_fn>

Options:
    -o FILE --output=FILE  Destination of primary output [default: stdout]
    -f FMT --format=FMT    Format of output, either csv or table [default: csv]
    --json=<json_fn>       Name of JSON output file
    --html=<html_fn>       Name of HTML output file
```

### `pca` - Principal component analysis

**class** de_toolkit.stats.**CountPCA**(*count_mat*)

Principal common analysis of the counts matrix.

This module performs PCA on a provided counts matrix and returns the principal component weights, scores, and variances. In addition, the weights and scores for each individual component can be combined to define the projection of each sample along that component.

The PCA module can also use a counts matrix that has associated column data information about the samples in each column. The user can specify some of these columns to include as variables for plotting purposes. The idea is that columns labeled with the same class will be colored according to their class, such that separations in the data can be more easily observed when projections are plotted.

**output**

Tabular output is a table where each row corresponds to a column in the counts matrix with the following fields:

**name** name of the column for the row

**PC*X*_*YY*** projections of principal component X (e.g. 1) that explains YY percent of the variance for each column

**properties**

Example JSON output:

```
[
    'name': 'pca',
    'stats': {
        'column_names': ['sample1','sample2',...],
        'column_variables': {
            'sample_names': ['sample1','sample2',...],
            'columns': [
                {
                    'column':'status',
                    'values':['disease','control',...]
                },
                {
                    'column':'batch',
                    'values':['b1','b1',...]
                },
        },
        'components': [
            {
                'name': 'PC1',
                'scores': [0.126,0.975,...], # length n
                'projections': [-8.01,5.93,...], # length m, ordered by
↪'column_names'
                'perc_variance': 0.75
            },
            {
                'name': 'PC2',
                'scores' : [0.126,0.975,...], # length n
                'projections': [5.93,-5.11,...], # length m
                'perc_variance': 0.22
            }
        ]
    }
]
```

Command line usage:

```
Usage:
    detk-stats pca [options] <counts_fn>

Options:
    -m FN --column-data=FN      Column data for annotating PCA results and
                                plots (experimental)
```

(continues on next page)

(continued from previous page)

```
    -f NAME --column-name=NAME  Column name from provided column data for
                                annotation PCA results and plots (experimental)
    -o FILE --output=FILE       Destination of primary output [default: stdout]
    -f FMT --format=FMT    Format of output, either csv or table [default: csv]
    --json=<json_fn>            Name of JSON output file
    --html=<html_fn>            Name of HTML output file
```

### `summary` - Common statistics set

de_toolkit.stats.**summary**(*count_mat*, *bins=20*, *log=False*, *density=False*)

Compute summary statistics on a counts matrix file.

This is equivalent to running each of these tools separately:

- basestats
- coldist
- colzero
- rowzero
- entropy
- pca

**Parameters**

- **count_mat** (*CountMatrix object*) – count matrix object
- **bins** (*int*) – number of bins, passed to coldist
- **log** (*bool*) – perform log10 transform of counts in coldist
- **density** (*bool*) – return a density distribution from coldist

**Returns** list of DetkModule subclasses for each of the called submodules

**Return type** list

Command line usage:

```
Usage:
    detk-stats summary [options] <counts_fn>

Options:
    -h --help
    --column-data=FN        Use column data provided in FN, only used in PCA
    --color-col=COLNAME     Use column data column COLNAME for coloring output plots
    --bins=BINS             Number of bins to use for the calculated
                            distributions [default: 20]
    --log                   log transform count statistics
    --density               Produce density distribution by dividing each distribution
                            by the appropriate sum
    -o FILE --output=FILE   Destination of primary output [default: stdout]
    -f FMT --format=FMT     Format of output, either csv or table [default: csv]
    --json=<json_fn>        Name of JSON output file
    --html=<html_fn>        Name of HTML output file
```

## 2.9 `transform` - Count Transformation


- *plog*
- *rlog*
- *vst*


Transformations of the distribution of counts in a matrix.

### 2.9.1 `plog`

de_toolkit.transform.**plog**(*count_obj*, *pseudocount=1*, *base=10*)

Logarithmic transform of a counts matrix with fixed pseudocount, i.e. $log(x+c)$

**Parameters** **count_obj** (*CountMatrix object*) – count matrix object

**Returns** log transformed counts dataframe with the same dimensionality as input counts

**Return type** pandas.DataFrame

Command line usage:

```
Usage:
    detk-transform plog [options] <count_fn>

Options:
    -c N --pseudocount=N   The pseudocount to use when taking the log transform␣
↪[default:1]
    -b B --base=B          The base of the log to use [default: 10]
    -o FILE --output=FILE  Destination of primary output [default: stdout]
```

### 2.9.2 `rlog`

Command line interface to the DESeq2 Regularized log (`rlog`) transformation. As in the originating package, the default behavior is to perform a blind transformation, i.e. without respect to an experimental design:

```
detk-transform rlog norm_counts.csv > rlog_norm_counts.csv
```

Roughly equivalent to the following R code:

```
library(DESeq2)

cnts <- as.matrix(read.csv("norm_counts.csv",row.names=1))
fakeColData <- # fake column data...

dds <- DESeqDataSetFromMatrix(countData = cnts,
    colData = fakeColData,
    design = ~ 1
)

dds <- rlog(dds,blind=True)
write.csv(assay(dds),out.fn)
```

To perform a non-blind transformation, a formula and column data file may be provided:

```
detk-transform rlog norm_counts.csv "counts ~ AgeOfDeath + Status" column_data.csv >␣
↪rlog_norm_counts_nonblind.csv
```

This invocation is roughly equivalent to the following R code:

```
library(DESeq2)

cnts <- as.matrix(read.csv("norm_counts.csv",row.names=1))
colData <- read.csv("column_data.csv",header=T,as.is=T,row.names=1)

dds <- DESeqDataSetFromMatrix(countData = cnts,
    colData = colData,
    design = ~ AgeOfDeath + Status
)

dds <- rlog(dds,blind=False)
write.csv(assay(dds),out.fn)
```

### 2.9.3 `vst`

Command line interface to the DESeq2 Regularized log (`vst`) transformation:

```
detk-transform vst norm_counts.csv > vst_norm_counts.csv
```

Roughly equivalent to the following R code:

```
library(DESeq2)

cnts <- as.matrix(read.csv("norm_counts.csv",row.names=1))
fakeColData <- # fake column data...

dds <- DESeqDataSetFromMatrix(countData = cnts,
    colData = fakeColData,
    design = ~ 1
)

dds <- vst(dds)
write.csv(assay(dds),out.fn)
```

## 2.10 `util` - Counts and Column Data File Utilities


- *tidy*
- *tidy-counts*
- *tidy-covs*


Functions for tidying up counts and column data files. Mostly this means subsetting one or the other so that the column IDs and order match. Combined with `csvgrep` from the csvkit package, this is useful for extracting subsets of samples for downstream differential expression analysis.

### 2.10.1 tidy

Subset both the counts columns and column data rows by intersection, returning new outputs for both. Note the tidied column data is not output by default, and the user must specify the `-p` argument to obtain it.

Command line usage:

```
Usage:
    detk-util tidy [options] <count_fn> <cov_fn>

Options:
    -o FILE --output=FILE  Destination of tidied counts data [default: stdout]
    -p FILE --column-data-output=FILE  Destination of tidied column data
```

### 2.10.2 tidy-counts

Subset and order the provided counts file columns according to the rows of the provided column data file. Operation will fail if there are rows in the column data file that do not exist as columns in the counts file.

Command line usage:

```
Usage:
    detk-util tidy-counts [options] <count_fn> <cov_fn>

Options:
    -o FILE --output=FILE  Destination of tidied counts data [default: stdout]
```

### 2.10.3 tidy-covs

Subset and order the provided column data file rows according to the columns of the provided ccounts data file. Operation will fail if there are columns in the counts file that do not exist as rows in the column data file.

Command line usage:

```
Usage:
    detk-util tidy-covs [options] <count_fn> <cov_fn>

Options:
    -o FILE --output=FILE  Destination of tidied column data [default: stdout]
```

## 2.11 Patsy-lite

### 2.11.1 Introduction

Some of the important operations in detk, most notably the `de` module, require the user specify a statistical model. The patsy python package is designed to describe statistical models in just this way. However, the syntax for describing patsy models is not very machine-readable, e.g. a linear model relating a continuous quantity to a categorical variable might look as follows:

```
cont_var ~ C(cat_var, levels=['A','B','C'])
```

The resulting full design matrix would have columns named something like:

```
cont_var ~ Intercept + C(cat_var, levels=['A','B','C'])[T.B] + C(cat_var, levels=['A',
↪'B','C'])[T.C]
```

This is at least a little bit yuck, and gets very yuck as the number of variables in the model increases. A more convenient and programmatically accessible way to express these models might be:

```
cont_var ~ cat_var[A,B,C]
```

resulting in the full matrix:

```
cont_var ~ Intercept + cat_var__B + cat_var__C
```

These variable names are much more amenable to programmatic use. Thus, detk implements a 'patsy-lite' syntax that follows these conventions, using the patsy library as the brain behind resolving full rank model matrices given column data.

### 2.11.2 Syntax

The patsy-lite syntax uses the column names from column data files passed to detk utilities. The following table contains example column data that will be useful in describing the patsy-lite syntax.

| cont | binary_str | binary_int | cat_str | cat_int |
|---|---|---|---|---|
| 0.13 | case | 1 | A | 1 |
| 0.97 | control | 0 | B | 2 |
| 0.22 | case | 1 | A | 1 |
| 0.76 | control | 0 | C | 3 |
| 0.69 | control | 0 | C | 3 |
| 0.08 | case | 1 | A | 1 |
| 0.17 | case | 1 | B | 2 |
| 0.53 | control | 0 | C | 3 |

Patsy-lite has four types of terms:

- *scalar*: variables that are to be considered purely numeric. Examples include continuous measures and ordinal variables, e.g. `cont`, `binary_int`, or `cat_int`.
- *binary*: variables that are binary categories or encoded as strings in the table. These terms are written in a design spec like `binary_str` or, if a reference group is specified, `binary_str[control]`. In this instance, the `control` value is reference group. Binary variables use a dummy encoding, such that only a single binary vector appears in the full model matrix.
- *multinomial*: categorical variables that have more than two levels. These terms are written in a design spec like `cat_str` or `cat_str[A,B,C,D]` to specify the desired order of levels. The first value specified (or by alphabetical order if omitted) is assumed to be the reference group. Multinomial variables use a dummy encoding, such that there is a binary vector for all but one of the levels in the full model matrix.
- *patsy*: there is limited support for passing other patsy term types (e.g. factorial terms, `binary_str:cat_str`). You can try to put in expressions that patsy knows how to understand, like `np.log(cont)`, but you do so at your own risk and I'm going to pretend to not be responsible for what befalls you if you choose to do so.

Every model is expected to have a set of terms on the left hand side and a set of terms on the right hand side separated by the ~ operator. In general, there should be only a single term on the left hand side. I didn't write any tests to see what would happen if there are more than one, so the usual disclaimer applies.

Depending on what type of term a variable is, the output column name in the full model matrix will follow one of three patterns:

- literal pass through - exactly the same as the column name
- categorical - `<variable name>__<level>`
- patsy-specific terms - e.g. `np.log(x)`

The double underscores in categorical variables should make it easy to recognize which variable an output column refers to, and make down-stream programmatic analysis easier.

Some mostly non-differential expression related examples, assuming the variable names in these examples are found in the appropriate column data file:

```
height ~ weight + age
disease_status[control] ~ age_at_death + batch + counts
gross_domestic_product ~ continent[NoAm] + population + election_year[no]
```

The full model matrix column names from these models might be something like:

```
height ~ Intercept + weight + age
disease_status__case ~ Intercept + age_at_death + batch__2 + batch__3 + counts
gross_domestic_product ~ Intercept + continent__SoAm + continent__Asia +
    continent__Aust + continent__Euro + continent__SoPo + population +
    election_year__yes
```

### 2.11.3 The special `counts` term

For most of the detk tools that use formulas, the intent is to perform some kind of differential expression where the feature counts are somewhere in the model. Since the patsy-lite terms refer to the column names in the column data file, the astute analyst might wonder how the counts term for each feature gets integrated into the formula. detk has a special term for this case: `counts`. This term can and should be included as if it was a normal *scalar* variable in the design spec.

The location of the `counts` variable is different depending on the DE method being used. For Firth's logistic regression, the counts variable is expected to be on the right hand side of the equation. For DESeq2, it is expected to be the only term on the left hand side. detk goes to less than Olympic lengths to ensure the design it gets has the `counts` variable in the place that it expects, but it should be ok.

## 2.12 `wrapr` - Thin wrapper for running R scripts

Thin wrapper interface for running R scripts from detk. This is a replacement for rpy2, which is a heavy dependency fraught with danger and hardship.

---

**Note:** This module is mostly intended for internal use by detk when interacting with R. A CLI interface is provided because why not, but is only intended to be used in advanced cases when you want to commandline-ize an R script that fits into the interface. If you have a one-off R script that needs to be integrated into your workflow, it would probably be better to just write it in R. Caveat emptor.

---

### 2.12.1 Setup

The `wrapr` interface assumes that R and any necessary packages have been already installed by the user. If you are using conda, you can install R easily with:

```
$ conda install -c r r-base
```

Once installed, `wrapr` also requires the jsonlite R package to be installed:

```
$ R
> install.packages("jsonlite")
```

To verify that your environment is properly set up to use `wrapr`, run:

```
$ detk wrapr check
R found: True
R path: /usr/bin/Rscript
jsonlite found: True
```

### 2.12.2 The interface

`wrapr` implements a well-defined interface between detk and R through a bridge script. From the command line, the following inputs are possible:

```
$ detk-wrapr run \
  --meta-in=/path/to/metadata \ # metadata filename corresponding to input counts
  --meta-out=/path/to/metadata_out \ # filename where modified metadata should be␣
↪written
  --params-in=/path/to/params.json \ # JSON formatted file with parameters needed by R
  --params-out=/path/to/output_params.json \ # filename where parameters/values can␣
↪be passed back out of R
  /path/to/rscript \ # R script written to use the interface
  /path/to/input_counts \ # counts matrix
  /path/to/output \ # filename where tabular output should be written
```

Arguments starting with `--` are optional. Input metadata and counts should be tabular as accepted elsewhere by detk. The input parameters file should be JSON formatted, containing an object with fields that are mapped directly to R `list` members.

The bridge script makes the following variables available in the R environment where the R script is run:

- **counts.fn**: path to the counts filename provided to `detk-wrapr`
- **out.fn**: path to the file where new counts will be written after R has operated on them, the user is expected to write to this file e.g. `write.csv(counts.mat, counts.out.fn)`
- **params**: an R `list` that contains parameter values as included in the parameter JSON file

For example, say we wanted to write an R script that added a configurable pseudocount to every count in a counts matrix. We could write the JSON parameter file as follows:

```
{
  "pseudocount": 1
}
```

And write the following R script, named `pseudocount.R`:

```
# counts.fn, params, and out.fn are already defined
mat <- read.csv(counts.fn,rownames=1,colnames=1)
new.mat <- mat + params$pseudocount
write.csv(new.mat,out.fn)
```

The command to execute this wrapr code might be:

```
$ detk wrapr run --params-in=pseudocount_params.json \
pseudocount.R counts.csv counts_plus_pseudo.csv
```

The file `counts_plus_pseudo.csv` will contain the result of the R script operation.

### 2.12.3 API documentation

**class** de_toolkit.wrapr.**WrapR**(*rscript_path*, *counts=None*, *metadata=None*, *params=None*, *output_fn=None*, *metadata_out_fn=None*, *params_out_fn=None*, *rpath=None*, *raise_on_error=True*, *routput_dir=None*)

Wrapper object for calling R code with Rscript.

**Note:** The attributes are only populated after the execute() method has been run

**Parameters**

- **rscript_path** (*str*) – path to the R script to run
- **counts** (*pandas.DataFrame, optional*) – dataframe containing counts to be passed to R
- **metadata** (*pandas.DataFrame, optional*) – dataframe containing metadata to be passed to R
- **params** (*dict, optional*) – dict of parameters to be passed to R
- **output_fn** (*str, optional*) – path to file where R should write output, if not provided the output is written to a temporary file and deleted upon WrapR object deletion
- **metadata_out_fn** (*str, optional*) – path to file where R should write metadata output
- **rpath** (*str*) – path to the Rscript executable, taken from the PATH environment variable if None
- **raise_on_error** (*bool*) – raise an exception if R encounters an error, other wise fail silently and deadly

**output**
*pandas.DataFrame* – dataframe of the tabular output created by R script

**metadata_out**
*pandas.DataFrame* – dataframe of the tabular metadata output created by R script

**params_out**
*dict* – dict of the output parameters list created by R script

**stdout**
*str* – string capturing the standard output of the R script

**stderr**
*str* – string capturing the standard error of the R script

**retcode**
*int* – return code of the R process

**success**
*bool* – True if retcode == 0

**Raises** `de_toolkit.wrapr.RExecutionError` – when *raise_on_error* is True, raise whenever R encounters an error

### Examples

Basic usage accepts a path to an R script and loads the content of the file pointed to by *out.fn* in the R script into the *output* attribute:

```
>>> with open('script.R','wt') as f :
        # note reference to implicitly defined *out.fn*
        # R variable
        f.write('write.csv(c(1,2,3,4),out.fn)')
>>> r = WrapR('script.R',output_fn='test.csv')
>>> r.execute()
>>> r.output
   x
1  1
2  2
3  3
4  4
>>> pandas.read_csv('test.csv',index_col=0)
   x
1  1
2  2
3  3
4  4
```

Can also use a context manager when the output doesn't need to be written to a named file:

```
>>> with WrapR('script.R') as r :
        r.execute()
        print(r.output)
   x
1  1
2  2
3  3
4  4
```

The standard output of the R script can be accessed with the *stdout* attribute:

```
>>> with open('euler.R','wt') as f :
        f.write('exp(complex(real=0,imag=pi))+1')
>>> with WrapR('euler.R','wt') as r :
        r.execute()
        print(r.stdout)
[1] 0+1.224647e-16i
```

de_toolkit.wrapr.**wrapr**(*Rcode*, ***kwargs*)

Convenience wrapper for WrapR object. Writes *Rcode* to a temporary file and executes it as it would if it were provided.

**Parameters** **Rcode** (*str*) – string containing valid R code to be executed

**Returns** A WrapR object executed with the code in input string

**Return type** obj

### Examples

```
>>> with wrapr('write.csv(c(1,2,3,4),out.fn)') as r :
        print(r.output)
   x
1  1
2  2
3  3
4  4
```

de_toolkit.wrapr.**get_r_path**()

Return the path to Rscript found in the shell environment.

de_toolkit.wrapr.**check_r**()

Tests whether the Rscript executable can be found.

de_toolkit.wrapr.**check_r_package**(*pkg*)

Tests whether the R package *pkg* is installed.

CHAPTER 3

# Installation

We suggest installing this package using pip:

```
pip install de_toolkit
```

In development or if you want to use the bleeding edge, when you want to run the toolkit, use the `setup.py` script:

```
python setup.py install
```

This should make the `detk` and its subtools available on the command line. Whenever you make changes to the code you will need to run this command again.

## 3.1 R dependencies

The following packages are only required to use the corresponding submodule functions:

- **R packages**
  - DESeq2 (`detk-de deseq2`, `detk-transform rlog`, `detk-transform vst`)
  - fgsea (`detk-enrich fgsea`)
  - logistf (`detk-de firth`)

We wearily suggest using anaconda to create an environment that contains the software necessary, e.g.:

```
conda create -n de_toolkit python=3.5

./install_conda_packages.sh

# if you want to use the R functions (Firth, DESeq2, etc.)
Rscript install_r_packages.sh
```

CHAPTER 4

# Indices and tables

- genindex
- modindex
- search

# Index


## B

BaseStats (class in de_toolkit.stats), 21

## C

check_r() (in module de_toolkit.wrapr), 38
check_r_package() (in module de_toolkit.wrapr), 38
ColDist (class in de_toolkit.stats), 21
ColZero (class in de_toolkit.stats), 23
CountPCA (class in de_toolkit.stats), 27

## E

Entropy (class in de_toolkit.stats), 26
entropy() (in module de_toolkit.outlier), 17

## F

fgsea() (in module de_toolkit.enrich), 12

## G

get_r_path() (in module de_toolkit.wrapr), 38

## M

metadata_out (de_toolkit.wrapr.WrapR attribute), 36

## O

output (de_toolkit.stats.BaseStats attribute), 21
output (de_toolkit.stats.ColDist attribute), 21
output (de_toolkit.stats.ColZero attribute), 23
output (de_toolkit.stats.CountPCA attribute), 28
output (de_toolkit.stats.Entropy attribute), 26
output (de_toolkit.stats.RowDist attribute), 23
output (de_toolkit.stats.RowZero attribute), 25
output (de_toolkit.wrapr.WrapR attribute), 36

## P

params_out (de_toolkit.wrapr.WrapR attribute), 36
plog() (in module de_toolkit.transform), 30
properties (de_toolkit.stats.ColDist attribute), 22
properties (de_toolkit.stats.ColZero attribute), 24
properties (de_toolkit.stats.CountPCA attribute), 28
properties (de_toolkit.stats.Entropy attribute), 26
properties (de_toolkit.stats.RowZero attribute), 25

## R

retcode (de_toolkit.wrapr.WrapR attribute), 37
RowDist (class in de_toolkit.stats), 23
RowZero (class in de_toolkit.stats), 25

## S

shrink() (in module de_toolkit.outlier), 18
stderr (de_toolkit.wrapr.WrapR attribute), 36
stdout (de_toolkit.wrapr.WrapR attribute), 36
success (de_toolkit.wrapr.WrapR attribute), 37
summary() (in module de_toolkit.stats), 29

## W

WrapR (class in de_toolkit.wrapr), 36
wrapr() (in module de_toolkit.wrapr), 37