Monod

Release 0.2

Gorin

May 02, 2023

CONTENTS

Monod implements procedures for fitting and analyzing stochastic models of transcription and sequencing.

The *Usage* page documents how to use the software, including its *Installation*.

For an example of the software applied to a small dataset, see the demo Colaboratory notebook.

Note: This project and its documentation are under active development.

CHAPTER

ONE

CONTENTS

1.1 Usage

1.1.1 Installation

To use *Monod*, install it from *pip*:

```
pip install monod
```

To use it in your code, import the package components:

```
import monod
from monod import *
from monod.preprocess import *
from monod.extract_data import *
from monod.cme_toolbox import CMEModel
from monod.inference import InferenceParameters, GradientInference
from monod.analysis import *
```

1.1.2 Quantification

To generate an intronic index for `kb`, obtain a reference genome and run `kb ref`:

```
kb ref -i ./refdata-gex-mm10-2020-A/kallisto/index.idx
-g ./refdata-gex-mm10-2020-A/t2g_mm10.txt
-f1 ./refdata-gex-mm10-2020-A/kallisto/cdna.fa
-f2 ./refdata-gex-mm10-2020-A/kallisto/intron.fa
-c1 ./refdata-gex-mm10-2020-A/kallisto/cdna_t2c.txt
-c2 ./refdata-gex-mm10-2020-A/kallisto/intron_t2c.txt
--workflow lamanno
./refdata-gex-mm10-2020-A/fasta/genome.fa
./refdata-gex-mm10-2020-A/genes/genes.gtf
```

To then generate the count matrices in `loom` format, use `kb count`:

```
kb count --verbose
-i ../ref/refdata-gex-mm10-2020-A/kallisto/index.idx
-g ../ref/refdata-gex-mm10-2020-A/t2g_mm10.txt
-x 10xv3
```

(continues on next page)

(continued from previous page)

```
-o OUTDIR/
-t 30 -m 30G
-c1 ../ref/refdata-gex-mm10-2020-A/kallisto/cdna_t2c.txt
-c2 ../ref/refdata-gex-mm10-2020-A/kallisto/intron_t2c.txt
--workflow lamanno --filter bustools --overwrite --loom
DATASET_FASTQ_LOCATIONS
```

This generates a `loom` file in `OUTDIR/counts_filtered/`.

1.1.3 Pre-processing

To define a "batch," or a set of inference runs, run `preprocess.construct_batch()`:

```
dir_string,dataset_strings = construct_batch(raw_filepaths, transcriptome_filepath,␣
↪dataset_names, \
                                             attribute_names, batch_location, meta,␣
↪batch_id, n_genes)
```

To import the files, specify the raw data (`loom`, `mtx`, or `adata`) filepaths in `raw_filepaths`, a gene length annotation filepath `transcriptome_filepath`, and various batch and dataset metadata. To specify the number of genes to analyze, set `n_genes`.

This will create a batch directory, dataset-specific subdirectories, the file `gene_set.csv` with the list of genes that meet filtering thresholds for all datasets, and the file `genes.csv` with the list of genes selected for further analysis. This gene list can also be defined or updated manually, but inference is typically unsuitable for genes that do not meet the filtering thresholds.

1.1.4 Model, data, and parameter definition

To define a CME model of transcription and sequencing, initialize an instance of `cme_toolbox.CMEModel`:

```
CMEModel(biological_model,sequencing_model)
```

where `biological_model = {'Bursty','Constitutive','Extrinsic','CIR'}` represents the transcriptional process and `sequencing_model = {'None','Bernoulli','Poisson'}` represents the dynamics of the sampling process.

To define the search parameters, initialize an instance of `inference.InferenceParameters`:

```
inference_parameters = InferenceParameters(phys_lb,phys_ub,samp_lb,samp_ub,gridsize,\
                    dataset_string,fitmodel,use_lengths)
```

where `phys_lb` and `phys_ub` are bounds on the transcriptional process model parameters, `samp_lb` and `samp_ub` are bounds on the sampling process model parameters, `gridsize` defines the grid for the sampling parameter scan, and `use_lengths` determines whether the unspliced mRNA capture rate depends on the gene length (to model priming at ubiquitous internal polyA sites).

To create a `SearchData` object to input into the inference process, run `extract_data.extract_data()`:

```
dir_string,dataset_strings = extract_data(loom_filepath, transcriptome_filepath, dataset_
↪name,
                                dataset_string, dir_string)
```

1.1.5 Running the inference pipeline

To run the pipeline, simply call the following parallelized code:

```
result_string = inference_parameters.fit_all_grid_points(n_cores,search_data)
```

This will iterate over all grid points using `n_cores` processors.

1.1.6 Post-processing and QC

To load the search results, import the file string:

```
sr = load_search_results(result_string)
```

To identify the technical noise parameter optimum, call a method of a SearchResults object *sr*:

```
sr.find_sampling_optimum()
```

Optionally, test its stability under subsampling and chi-squared testing:

```
fig1,ax1 = plt.subplots(1,1)
sr.plot_landscape(ax1)
_=sr.chisquare_testing(sd)
sr.resample_opt_viz()
sr.resample_opt_mc_viz()
sr.chisq_best_param_correction(sd,viz=True)
```

Optionally, examine whether the distribution fits match the raw data:

```
sr.plot_gene_distributions(sd,marg='joint')
sr.plot_gene_distributions(sd,marg='nascent')
sr.plot_gene_distributions(sd,marg='mature')
```

To chracterize the uncertainty, variation, and bias in biological parameters, compute the standard errors of their maximum likelihood estimates, then plot their distributions and dependence on length (which should be minimal):

```
sr.compute_sigma(sd,num_cores)
sr.plot_param_L_dep(plot_errorbars=True,plot_fit=True)
sr.plot_param_marg()
```

As the standard error computation is typically computationally intensive, it is useful to store an updated copy on disk after evaluating it:

```
sr.update_on_disk()
```

1.1.7 Noise decomposition

Two complementary methods are available for investigating the contributions of different noise sources. The first is non-parametric; calling a method of a `SearchData` object `sd` returns the fractions of variation retained and discarded after normalization and log-transformation:

```
f = sd.get_noise_decomp()
```

These fractions are not guaranteed to be positive, because the transformations may *increase* the relative spread of the data. On the other hand, if a fit has been completed, a method of a `SearchResults` object `sr` reports the fractions of intrinsic, extrinsic (bursty), and technical variation for each gene and molecular species:

```
f = sr.get_noise_decomp()
```

1.1.8 Differential parameter value identification

Given a set of matched datasets, run with the same model over the same set of genes, two approaches are available for identifying putative patterns of differential expression and regulation. A moment-based, biology-agnostic one uses a simple *t*-test to identify differences in the means of spliced counts in `SearchData` objects `sd1` and `sd2`:

```
gf = compute_diffexp(sd1,sd2)
```

where `gf` is boolean vector that reports `True` if the gene is identified as DE. However, this approach cannot identify differences if biological parameters change in a correlated way and the mean stays the same. We introduce a more mechanistic approach for the identification of differential expression suggested by parameter variation, based on two `SearchResults` objects `sr1` and `sr2`:

```
gf = compute_diffreg(sr1,sr2)
```

where `gf` is a two-dimensional boolean array that reports `True` if a particular *parameter* is identified as DE. After using these arrays to find a subpopulation of interest – e.g., genes that do not exhibit variation in the spliced mean, but do exhibit modulation in the burst size – it is possible to plug the gene filter `genes_to_plot` back in to inspect the raw data and fits:

```
gf = compare_gene_distributions(sr_arr,sd_arr,genes_to_plot=genes_to_plot)
```

1.2 API

The API is currently under construction, but comments are available for all of the functions in *Monod* `py` files.