mpathic Documentation

Release 0.0.1

Ammar Tareen, William Ireland, Justin Kinney

Feb 04, 2019

Contents

CHAPTER 1

Quantitative Modeling of Sequence-function Relationships for Massively Parallel Assays

Written by Ammar Tareen

MPAthic[1] is a python API and it infers quantitative models from data. Most MPAthic classes take in one or more tabular text files as input and return a tabular text file as output. All input and output files are designed to be human readable. The first line of each tabular text file contains headers describing the contents of each column. All input files are required to have the proper set of columns, which of course depend on the command being executed. By default, input is taken from the standard input and output is written to the standard output.

[1] William T. Ireland and Justin B. Kinney (2016) MPAthic: quantitative modeling of sequence-function relationships for massively parallel assays PDF.

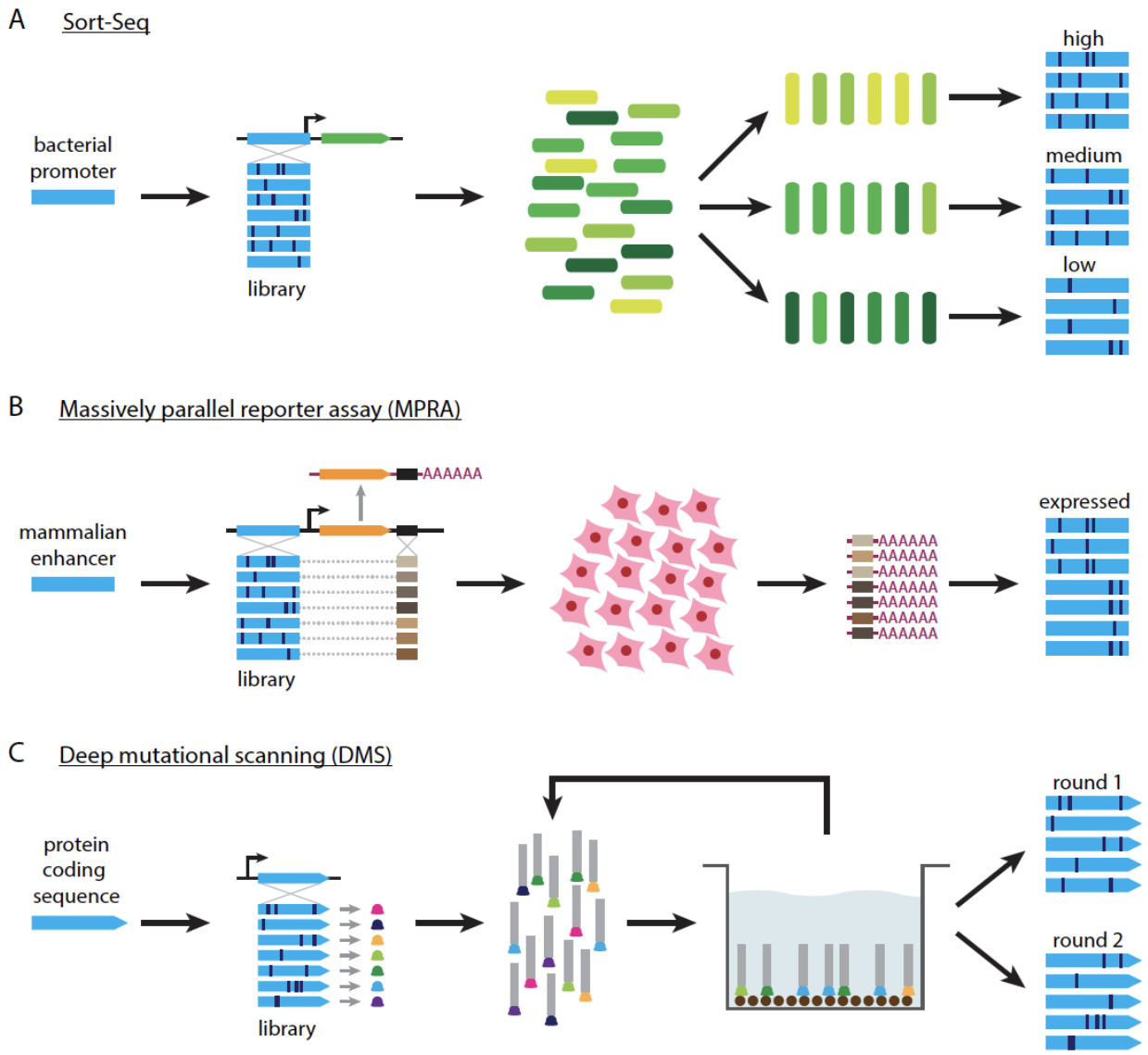

Figure 1

Figure 1 Three different massively parallel experiments. (A) The Sort-Seq assay of [7]. A plasmid library is generated in which mutagenized versions of a bacterial promoter (blue) drive the expression of a fluorescent protein (green). Cells carrying these plasmids are then sorted according to measured fluorescence using fluorescence-activated cell sorting (FACS). The variant promoters in each bin of sorted cells are then sequenced. (B) The MPRA assay of [8]. Variant enhancers (blue) are used to drive the transcription of RNA that contains enhancer-specific tags (shades of brown). Expression constructs are transfected into cell culture, after which tag-containing RNA is isolated and sequenced. Output sequences consist of the variant enhancers that correspond to expressed tags. (C) The DMS assay of [9]. Randomly mutagenized gene sequences (blue) produce variant proteins (colored bells) that are expressed on the surface of phage (gray rectangles). Panning is used to enrich for phage that express proteins that bind a specific ligand of interest (brown circles). The variant coding regions enriched after one or more rounds of panning are then sequenced.

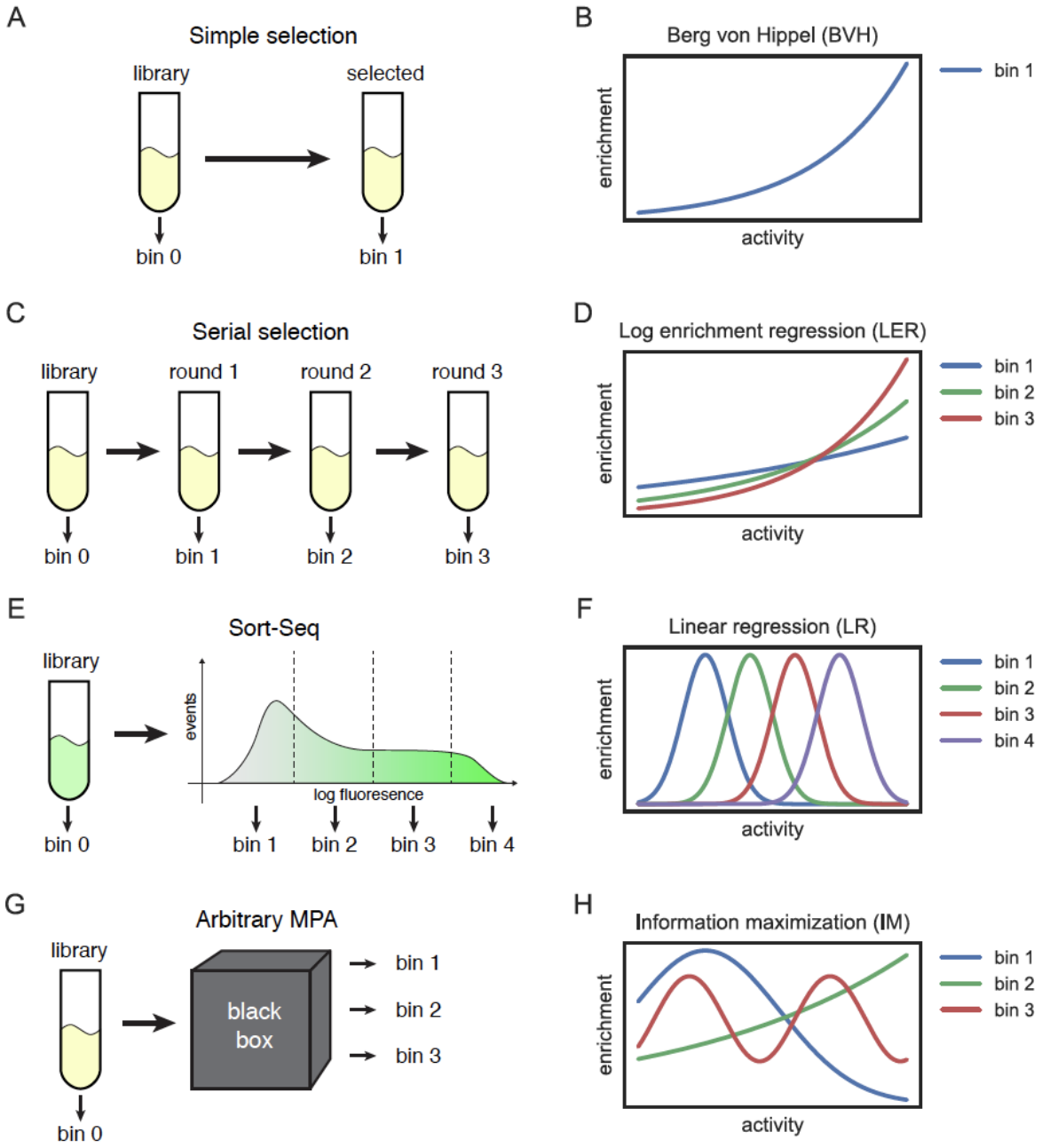

Figure 4

Figure 4 Different model fitting methods are needed for different situations. (A) In a simple selection experiment, sequences in a library are enriched in for the activity of interest. (B) If enrichment is an exponential function of sequence activity (as is often assumed), matrix modeling using the method of Berg and von Hippel (BVH) is justified. BVH inference, however, can be applied only to datasets consisting of two bins. (C) In a serial selection experiment, the sequences in different bins experience progressively strong enrichment. (D) If each step performs exponential exponential, then the enrichment in each bin will be a different exponential function of activity. In such cases, log enrichment regression (LER) provides a sensible way to infer models using data from all bins. (E) In Sort-Seq experiments, variants are often partitioned using gates that are evenly spaced in log fluorescence. (F) Such sorting can cause the enrichment in each bin to be an approximately Gaussian function of activity. In this case, inference using linear regression (LR) is justified. (G) All massively parallel assays can be thought of as a black box that takes sequences from a library and outputs to these sequences to a small number of bins. (H) Information maximization (IM), unlike BVH, LER, and LR, can be applied to MPA data regardless of how enrichment depends on sequence activity. In B,D,F,H, the "enrichment" for bin M is a function of sequence activity that is given by $p(\text{activity}|\text{bin } M)/p(\text{activity}|\text{bin } 0)$.

1.1 Installation

1.1.1 Prerequisites

MPAthic uses some Non-python resources and has the following prerequisites for installation:

- **GCC** GCC, which contains both C compilers and Fortran compilers, is required to run MPAthic. From with in terminal, run the command gcc.

```
$ gcc
$ clang: error: no input files
```

  Check that gfortran was installed with gcc.

```
$ gfortran
$ gfortran: fatal error: no input files ...
```

- **NUMPY** numpy is required to be installed before mpathic can be installed.

```
pip install numpy
```

1.1.2 Install MPAthic

With the prerequisites installed, MPAthic can be installed using the pip (version 9.0.0 or higher) from PyPI. At the command line:

```
pip install mpathic
```

The code for MPAthic is open source and available on GitHub. Some commonly encountered installation issues can be found here: *Common Installation Issues*

1.2 Quick Start

The following snippets show how to use MPAthic from within python.

```
import mpathic as mpa
mpa.demo()
```

1.3 Resources

1.3.1 Tutorial

Import the MPAthic package as follows:

```
import mpathic as mpa
```

Simulating Data

We begin by simulating a library of variant CRP binding sites. We can use the *mpa.SimulateLibrary* class to create a library of random mutants from an initial wildtype sequence and mutation rate:

```
sim_library = mpa.SimulateLibrary(wtseq="TAATGTGAGTTAGCTCACTCAT", mutrate=0.24)
sim_library.output_df.head()
```

The *output_df* attribute of the `sim_library` class looks like the dataframe below

ct	seq
21	TAATGTGAGTTAGCTCACTCAT
7	TAATGTGAGTTAGCTAACTCAT
6	TAATGTGAGTTAGCTCACTCAA

1	TAATGTGTGTTCGCTCATCCAT

In general, MPAthic datasets are pandas dataframes, comprising of columns of counts and sequence values. To simulate a Sort-Seq experiment ([1]), we use the *mpa.SimulateSort* class. This class requires a dataset input and a model dataframe input. We first import these inputs using `io` module provided with the MPAthic package:

```
# Load dataset and model dataframes
dataset_df = mpa.io.load_dataset('sort_seq_data.txt')
model_df = mpa.io.load_model('crp_model.txt')
```

Next, we call the `SimulateSort` class as follows:

```
# Simulate a Sort-Seq experiment
sim_sort = mpa.SimulateSort(df=dataset_df,mp=model_df)
sim_sort.output_df.head()
```

The head of the output dataframe looks like

ct	ct_0	ct_1	ct_2	ct_3	ct_4	seq
4	0	0	1	0	0	AAAAAAGGTGAGTTAGCTAACT
3	0	0	0	0	1	AAAAAATATAAGTTAGCTCGCT
4	0	0	0	1	0	AAAAAATATGATTTAGCTGACT
3	0	0	0	0	1	AAAAAATGTCAGTTAGCTCACT
4	0	0	1	0	0	AAAAAATGTGAATTATCGCACT

Computing Profiles

It is often useful to compute the mutation rate within a set of sequences, e.g., in order to validate the composition of a library. This can be accomplished using the *mpa.ProfileMut* class as follows:

```
profile_mut = mpa.ProfileMut(dataset_df = dataset_df)
profile_mut.mut_df.head()
```

The mutation rate at each position within the sequences looks like

[1] Kinney JB, Anand Murugan, Curtis G. Callan Jr., and Edward C. Cox (2010) Using deep sequencing to characterize the biophysical mechanism of a transcriptional regulatory sequence. PNAS May 18, 2010. 107 (20) 9158-9163; PDF.

pos	wt	mut
0	A	0
1	A	0
2	A	0.33871
3	T	0.127566
4	A	0.082111

To view the frequency of occurrence for every base at each position, use the *mpa.ProfileFreq* class:

```
profile_freq = mpa.ProfileFreq(dataset_df = dataset_df)
profile_freq.freq_df.head()
```

pos	freq_A	freq_C	freq_G	freq_T
0	1	0	0	0
1	1	0	0	0
2	0.66129	0.33871	0	0
3	0.043988	0.042522	0.041056	0.872434
4	0.917889	0.019062	0.02566	0.03739

Information pro les (also called "information footprints") provide a particularly useful way to identify functional positions within a sequence. These pro les list, for each position in a sequence, the mutual information between the character at that position and the bin in which a sequence is found. Unlike mutation and frequency profiles, which require sequence counts for a single bin only, information profiles are computed from full datasets, and can be accomplished using the *mpa.ProfileInfo* class as follows:

```
profile_info = mpa.ProfileInfo(dataset_df = dataset_df)
profile_info.info_df.head()
```

pos	info
0	0.000077
1	0.000077
2	0.008357
3	0.008743
4	0.013745

Quantitative Modeling

The *mpa.LearnModel* class can be used to fit quantitative models to data:

```
learned_model = mpa.LearnModel(df=dataset_df)
learned_model.output_df.head()
```

pos	val_A	val_C	val_G	val_T
0	-0.201587	0.067196	0.067196	0.067196
1	-0.201587	0.067196	0.067196	0.067196
2	-0.10637	-0.167351	0.13686	0.13686
3	-0.287282	0.041222	-0.2039	0.44996
4	-0.056109	-0.871858	0.344537	0.583429

The purpose of having a quantitative model is to be able to predict the activity of arbitrary sequences. This basic operation is accomplished using the *mpa.EvaluateModel* class:

```
eval_model = mpa.EvaluateModel(dataset_df = dataset_df, model_df = model_df)
eval_model.out_df.head()
```

ct	ct_0	ct_1	ct_2	ct_3	ct_4	seq	val
1	0	0	1	0	0	AAAGGTGAGTTAGCTAACTCAT	0.348108
1	0	0	0	0	1	AAATATAAGTTAGCTCGCTCAT	-0.248134
1	0	0	0	1	0	AAATATGATTTAGCTGACTCAT	0.009507
1	0	0	0	0	1	AAATGTCAGTTAGCTCACTCAT	0.238852
1	0	0	1	0	0	AAATGTGAATTATCGCACTCAT	-0.112121

Often, it is useful to scan a model over all sequences embedded within larger contigs. To do this, MPAthic provides the class *mpa.ScanModel*, which is called as follows:

```
# get contigs, provided with mpathic
fastafile = "./mpathic/examples/genome_ecoli_1000lines.fa"
contig = mpa.io.load_contigs_from_fasta(fastafile, model_df)

scanned_model = mpa.ScanModel(model_df = model_df, contigs_list = contigs_list)
scanned_model.sitelist_df.head()
```

	val	seq	left	right	ori	contig
0	2.040628	GGTCGTTTGCC[illegible]CGCCGTGC[illegible]	11710	11731	•	MG1655.fa
1	2.00608	GGAAGTCGCC[illegible]CGCACCG[illegible]	74727	74748	•	MG1655.fa
2	1.996992	TGGGTGTGGC[illegible]TGACCTG[illegible]	45329	45350	•	MG1655.fa
3	1.920821	GGTATGTGTC[illegible]AGCCAGG[illegible]	38203	38224	•	MG1655.fa
4	1.879852	GGTGATTTTG[illegible]TGGTGGC[illegible]	73077	73098	•	MG1655.fa

A good way to assess the quality of a model is to compute its predictive information on a massively parallel data set. This can be done using the *predictive_info* (need to write this) class:

```
predictive_info = mpa.PredictiveInfo(data_df = dataset_df, model_df = model_df,
↪start=52)
```

References

1.3.2 Examples

Simulations

```
# Simulate library example

import mpathic as mpa
```

(continues on next page)

(continued from previous page)

```
# reate a library of random mutants from an initial wildtype sequence and mutation␣
↪rate
sim_library = mpa.SimulateLibrary(wtseq="TAATGTGAGTTAGCTCACTCAT", mutrate=0.24)
sim_library.output_df.head()

# Load dataset and model dataframes
dataset_df = mpa.io.load_dataset('sort_seq_data.txt')
model_df = mpa.io.load_model('crp_model.txt')

# Simulate a Sort-Seq experiment example
sim_sort = mpa.SimulateSort(df=dataset_df,mp=model_df)
sim_sort.output_df.head()
```

Profiles

```
import mpathic as mpa


# Load dataset and model dataframes
dataset_df = mpa.io.load_dataset('sort_seq_data.txt')
model_df = mpa.io.load_model('crp_model.txt')

# mut profile example
profile_mut = mpa.ProfileMut(dataset_df = dataset_df)
profile_mut.mut_df.head()

# freq profile example
profile_freq = mpa.ProfileFreq(dataset_df = dataset_df)
profile_freq.freq_df.head()

# info profile example
profile_info = mpa.ProfileInfo(dataset_df = dataset_df)
profile_info.info_df.head()
```

Models

```
import mpathic as mpa

# Load dataset and model dataframes
dataset_df = mpa.io.load_dataset('sort_seq_data.txt')
model_df = mpa.io.load_model('crp_model.txt')

# learn models example
learned_model = mpa.LearnModel(df=dataset_df)
learned_model.output_df.head()

# evaluate models example
eval_model = mpa.EvaluateModel(dataset_df = dataset_df, model_df = model_df)
eval_model.out_df.head()

# scan models example
# get contigs, provided with mpathic
fastafile = "./mpathic/examples/genome_ecoli_1000lines.fa"
contig = mpa.io.load_contigs_from_fasta(fastafile, model_df)

scanned_model = mpa.ScanModel(model_df = model_df, contigs_list = contigs_list)
```

(continues on next page)

(continued from previous page)

```
scanned_model.sitelist_df.head()

# predictive info example
predictive_info = mpa.PredictiveInfo(data_df = dataset_df, model_df = model_df,
↪start=52)
```

1.3.3 Documentation

mpa.SimulateLibrary

Overview

simulate library is a program within the mpathic package which creates a library of random mutants from an initial wildtype sequence and mutation rate.

Usage

```
>>> import mpathic
>>> mpathic.SimulateLibrary(wtseq="TAATGTGAGTTAGCTCACTCAT")
```

Example Output Table:

```
ct              seq
1002            TAATGTGAGTTAGCTCACTCAT
50              TAATGTGAGTTAGATCACTCAT
...
```

Class Details

class simulate_library.**SimulateLibrary**(***kwargs*)

Parameters

wtseq [(string)] wildtype sequence. Must contain characteres 'A', 'C', 'G','T' for dicttype = 'DNA', 'A', 'C', 'G','U' for dicttype = 'RNA'

mutrate [(float)] mutation rate.

numseq [(int)] number of sequences. Must be a positive integer.

dicttype [(string)] sequence dictionary: valid choices include 'dna', 'rna', 'pro'

probarr [(np.ndarray)] probability matrix used to generate bases

tags [(boolean)] If simulating tags, each generated seq gets a unique tag

tag_length [(int)] Length of tags. Should be >= 0

Attributes

output_df [(pandas dataframe)] Contains the output of simulate library in a pandas dataframe.

arr2seq(*arr*, *inv_dict*)

Change numbers back into base pairs.

seq2arr (*seq*, *seq_dict*)
Change base pairs to numbers

mpa.SimulateSort

Contents

Overview

`SimulateSort` is a program within the mpathic package which simulates performing a Sort Seq experiment.

Usage

```
>>> import mpathic
>>> loader = mpathic.io
>>> mp_df = loader.load_model('./mpathic/examples/true_model.txt')
>>> dataset_df = loader.load_dataset('./mpathic/data/sortseq/full-0/library.txt')
>>> mpathic.SimulateSort(df=dataset_df,mp=mp_df)
```

Example Input and Output

The input table to this function must contain sequence, counts, and energy columns

Example Input Table:

```
seq    ct    val
AGGTA  5     -.4
AGTTA  1     -.2
...
```

Example Output Table:

```
seq    ct    val    ct_1     ct_2     ct_3 ...
AGGTA  5     -.4    1        2        1
AGTTA  1     -.2    0        1        0
...
```

The output table will contain all the original columns, along with the sorted columns (ct_1, ct_2 . . .)

Class Details

class simulate_sort.**SimulateSort** (***kwargs*)
Simulate cell sorting based on expression.

Parameters

df: (pandas dataframe) Input data frame.

mp: (pandas dataframe) Model data frame.

noisetype: (string, None) Noise parameter string indicating what type of noise to include. Valid choices include None, 'Normal', 'LogNormal', 'Plasmid'

npar: (list) parameters to go with noisetype. E.g. for

noisetype 'Normal', npar must contain the width of the normal distribution

nbins: (int) Number of bins that the different variants will get sorted into.

sequence_library: (bool) A value of True corresponds to simulating sequencing the library in bin zero

start: (int) Position to start analyzed region

end: (int) Position to end analyzed region

chunksize: (int) This represents the size of chunk the data frame df will be traversed over.

Attributes

output_df: (pandas data frame) contains the output of the simulate_sort constructor

mpa.ProfileFreq

Overview

`ProfileFreq` is a program within the mpathic package which calculates the fractional occurrence of each base or amino acid at each position.

Usage

```
>>> import mpathic as mpa
>>> mpa.ProfileFreq(dataset_df = dataset_df)
```

Example Input and Output

Input tables must contain a position column (labeled ''pos'') and columns for each base or amino acid (labeled ct_A, ct_C...).

Example Input Table:

```
pos ct_A ct_C ct_G ct_T
0   10   20   40   30
...
```

Example Output Table:

```
pos freq_A freq_C freq_G freq_T
0   .1     .2     .4     .3
...
```

Class Details

class profile_freq.**ProfileFreq**(***kwargs*)

Profile Frequencies computes character frequencies (0.0 to 1.0) at each position

Parameters

dataset_df: (pandas dataframe) A dataframe containing a valid dataset.

bin: (int) A bin number specifying which counts to use

start: (int) An integer specifying the sequence start position

end: (int) An integer specifying the sequence end position

Returns

freq_df: (pd.DataFrame) A dataframe containing counts for each nucleotide/amino acid character at each position.

mpa.ProfileMut

Overview

It is often useful to compute the mutation rate within a set of sequences, e.g., in order to validate the composition of a library. This can be accomplished using the profile mut class as follows:

Usage

```
>>> import mpathic as mpa
>>> mpa.ProfileMut(dataset_df = valid_dataset)
```

Example Input:

```
ct                    seq

259  TAATGTGAGTTAGCTCACTCAT
41  TAAAGTGAGTTAGCTCACTCAT
36  TAATGTGAGTAAGCTCACTCAT
35  TAGTGTGAGTTAGCTCACTCAT
34  TAATGTTAGTTAGCTCACTCAT
34  TTATGTGAGTTAGCTCACTCAT
...
```

Example Output:

```
    pos wt      mut

0     0  T  0.23819
1     1  A  0.24141
2     2  A  0.24118
3     3  T  0.24016
4     4  G  0.24093
5     5  T  0.24001
...
```

Class Details

class profile_mut.**ProfileMut**(***kwargs*)

Parameters

dataset_df: (pandas dataframe) Input data frame containing a valid dataset.

bin: (int) A bin number specifying which counts to use

start: (int) An integer specifying the sequence start position

end: (int) An integer specifying the sequence end position

err: (boolean) If true, include error estimates in computed mutual information

Returns

mut_df: (pandas data frame) A pandas dataframe containing results.

mpa.ProfileInfo

Overview

`profile_info` is a program within the mpathic package which calculates the mutual information between base identity at a given position and expression for each position in the given data set.

Usage

```
>>> import mpathic as mpa
>>> mpa.ProfileInfo(dataset_df = dataset_df)
```

Example Input and Output

The input to the function must be a sorted library a column for sequences and columns of counts for each bin. For selection experiments, ct_0 should label the pre-selection library and ct_1 should be the post selection library. For MPRA experiments, ct_0 should label the sequence library counts, and ct_1 should label the mRNA counts.

Example input table:

```
seq       ct_0     ct_1      ct_2...
ACATT     1        4         3
GGATT     2        5         5
...
```

Example output table:

```
pos     info     info_err
0       .02      .004
1       .04      .004
...
```

The mutual information is given in bits.

Class Details

class profile_info.**ProfileInfo**(***kwargs*)

Profile Info computes the mutual information (in bits), at each position, between the character and the bin number.

Parameters

dataset_df: (pandas dataframe) Input data frame

err: (boolean) If true, include error estimates in computed mutual information

method: (string) method used in computation. Valid choices inlcude: 'naive','tpm','nsb'.

pseudocount: (float) pseudocount used to compute information values

start: (int) An integer specifying the sequence start position

end: (int) An integer specifying the sequence end position

Returns

info_df: (pandas dataframe) dataframe containing results.

mpa.LearnModel

Contents

Overview

`LearnModel` is a program within the mpathic package which generates linear energy matrix models for sections of a sorted library.

Usage:

```
>>> import mpathic
>>> loader = mpathic.io
>>> filename = "./mpathic/data/sortseq/full-0/data.txt"
>>> df = loader.load_dataset(filename)
>>> mpathic.LearnModel(df=df,verbose=True,lm='ER')
```

Example Input and Output

There are two types of input dataframes learn model can accept as input: Matrix models and neighbour models. The input table to this program must contain a sequences column and counts columns for each bin. For a sort seq experiment, this can be any number of bins. For MPRA and selection experiments this must be ct_0 and ct_1.

Matrix models Input Dataframe:

```
seq        ct_0        ct_1        ct_2        ct_3        ct_4

AAAAAAGGTGAGTTA   0.000000   0.000000   1.000000   0.000000   0.000000
AAAAAATATAAGTTA   0.000000   0.000000   0.000000   0.000000   1.000000
AAAAAATATGATTTA   0.000000   0.000000   0.000000   1.000000   0.000000
...
```

Neighbour Model:

```
pos      val_AA      val_AC      val_AG      val_AT      val_CA      val_CC      val_CG
↪val_CT      val_GA      val_GC      val_GG      val_GT      val_TA      val_TC      val_TG
↪    val_TT
  0   0.081588  -0.019021   0.007188   0.042818  -0.048443  -0.015712  -0.053949  -0.
↪024360  -0.025149  -0.030791  -0.022920  -0.026910   0.052324   0.002189  -0.014354
↪  0.095505
  1   0.033288  -0.005410   0.014198   0.018246  -0.033583  -0.001761  -0.020431  -0.
↪007561  -0.018550  -0.025738  -0.028961  -0.010787   0.007764   0.024888  -0.000199
↪  0.054599
  2  -0.026142   0.008002  -0.029641   0.036698  -0.001028  -0.008025  -0.022645   0.
↪023678   0.006907  -0.016295  -0.054918   0.028913  -0.005400   0.003121   0.000996
↪  0.055780
```

(continues on next page)

(continued from previous page)

```
  3  -0.046159  -0.006071  -0.001542   0.028109  -0.020442  -0.024574   0.056595  -0.
↪024776  -0.005172  -0.055010  -0.029327  -0.016699   0.001295  -0.016304   0.128112␣
↪  0.031967
 ...
```

Example Output Table:

```
pos     val_A     val_C     val_G     val_T
0     0  0.000831 -0.014006  0.144818 -0.131643
1     1 -0.033734  0.087419 -0.029997 -0.023688
2     2  0.009189  0.018999  0.026719 -0.054908
3     3 -0.003516  0.073503  0.001759 -0.071745
4     4  0.062168 -0.028879 -0.057249  0.023961
...
```

Class Details

class learn_model.**LearnModel**(***kwargs*)

Constructor for the learn model class. Models can be learnt via the matrix model or the neighbor model. Matrix models assume independent contributions to activity from characters at a particular position whereas neighbor model assume near contributions to activity from all possible adjacent characters.

Parameters

df: (pandas data frame) Dataframe containing several columns representing bins and sequence column. The integer values in bins represent the occurrence of the sequence that bin.

lm: (str) Learning model. Possible values include {'ER','LS','IM', 'PR'}. 'ER': enrichment ratio inference. 'LS': least squares optimization. 'IM' : mutual information maximization (similar to maximum likelihood inference in the large data limit). 'PR' stands for Poisson Regression.

modeltype: (string) Type of model to be learned. Valid choices include "MAT" and "NBR", which stands for matrix model and neigbhour model, respectively. Matrix model assumes mutations at a location are independent and neighbour model assumes epistatic effects for mutations.

LS_means_std: (pandas dataframe) For the least-squares method, this contains the user supplied mean and standard deviation. The order of the columns is ['bin', 'mean', 'std'].

db: (string) File name for a SQL script; it could be passed in to the function MaximizeMI_memsaver

iteration: (int) Total number of MCMC iterations to do. Passed in the sample method from MCMC.py which may be part of pymc.

burnin: (int) Variables will not be tallied until this many iterations are complete (thermalization).

thin: (int) Similar to parameter burnin, but with smaller default value.

runnum: (int) Run number, used to determine the correct sql script extension in MaximizeMI_memsaver

initialize: (string) Variable for initializing the learn model class constructor. Valid values include "rand", "LS", "PR". rand is MCMC, LS is least squares and PR and poisson regression.

start: (int) Starting position of the sequence.

end: (int) end position of the sequence.

foreground: (int) Indicates column number representing foreground (E.g. can be passed to Berg_Von_Hippel method).

background: (int) Indicates column number representing background.

alpha [(float)] Regularization strength; must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization. Alpha corresponds to `C^-1` in other linear models such as LogisticRegression or LinearSVC. (this snippet taken from ridge.py written by Mathieu Blondel)

pseudocounts: (int) A artificial number added to bin counts where counts are really low. Needs to be Non-negative.

verbose: (bool) A value of false for this parameter suppresses the output to screen.

tm: (int) Number bins. DOUBLE CHECK.

Berg_von_Hippel (*df*, *dicttype*, *foreground=1*, *background=0*, *pseudocounts=1*)
Learn models using berg von hippel model. The foreground sequences are usually bin_1 and background in bin_0, this can be changed via flags.

Compute_Least_Squares (*raveledmat*, *batch*, *sw*, *alpha=0*)
Ridge regression is the only sklearn regressor that supports sample weights, which will make this much faster

Markov (*df*, *dicttype*, *foreground=1*, *background=0*, *pseudocounts=1*)
Learn models using berg von hippel model. The foreground sequences are usually bin_1 and background in bin_0, this can be changed via flags.

MaximizeMI_memsaver(*seq_mat*, *df*, *emat_0*, *wtrow*, *db=None*, *burnin=1000*, *iteration=30000*, *thin=10*, *runnum=0*, *verbose=False*)
Performs MCMC MI maximzation in the case where lm = memsaver

find_second_NBR_matrix_entry(*s*)
this is a function for use with numpy apply along axis. It will take in a sequence matrix and return the second nonzero entry

weighted_std(*values*, *weights*)
Takes in a dataframe with seqs and cts and calculates the std

mpa.EvaluateModel

Contents

Overview

`EvaluateModel` can be used to predict the activity of arbitrary sequences.

Usage

```
>>> import mpathic as mpa
>>> model = mpa.io.load_model("./mpathic/data/sortseq/full-0/crp_model.txt")
>>> dataset = mpa.io.load_dataset("./mpathic/data/sortseq/full-0/data.txt")
>>> mpa.EvaluateModel(dataset_df = dataset, model_df = model)
```

Example Input and Output

Example Input Table:

```
pos      val_A      val_C      val_G      val_T
3  -0.070101  -0.056502   0.184170  -0.057568
4  -0.045146  -0.042017   0.172377  -0.085214
5  -0.035447   0.006974   0.059453  -0.030979
6  -0.037837  -0.000299   0.079747  -0.041611
7  -0.110627  -0.054740   0.066257   0.099110
...
```

Example Output Table:

```
output:

 0         0.348108
 1        -0.248134
 2         0.009507
 3         0.238852
 4        -0.112121
 5        -0.048588
...
```

Class Details

class evaluate_model.**EvaluateModel**(***kwargs*)

Parameters

dataset_df: (pandas dataframe) Input dataset data frame

model_df: (pandas dataframe) Model dataframe

left: (int) Seq position at which to align the left-side of the model.

Defaults to position determined by model dataframe.

right: (int) Seq position at which to align the right-side of the model.

Defaults to position determined by model dataframe.

mpa.ScanModel

Overview

The scan model class can scan a model over all sequences embedde within larger contigs.

Usage

```
>>> import mpathic as mpa
>>> model = mpa.io.load_model("./mpathic/data/sortseq/full-0/crp_model.txt")
>>> fastafile = "./mpathic/examples/genome_ecoli_1000lines.fa"
>>> contig = mpa.io.load_contigs_from_fasta(fastafile,model)
>>> mpa.ScanModel(model_df = model, contig_list = contig)
```

Example Output Table:

```
    val                    seq   left  right ori     contig
0  2.040628  GGTCGTTTGCCTGCGCCGTGCA  11710  11731   +  MG1655.fa
1  2.006080  GGAAGTCGCCGCCCGCACCGCT  74727  74748   -  MG1655.fa
2  1.996992  TGGGTGTGGCGCGTGACCTGTT  45329  45350   +  MG1655.fa
3  1.920821  GGTATGTGTCGCCAGCCAGGCA  38203  38224   +  MG1655.fa
4  1.879852  GGTGATTTTGGCGTGGTGGCGT  73077  73098   -  MG1655.fa
5  1.866188  GTTCTTTTCCGCGGGCTGGGAT  35967  35988   -  MG1655.fa
...
```

Class Details

class scan_model.**ScanModel**(*model_df*, *contig_list*, *numsites=10*, *verbose=False*)

Parameters

model_df: (pandas dataframe) The dataframe containing a model of the binding energy and a wild type sequence.

contig_list: (list) list containing contigs. Can be loaded from fasta file via mpathic.io.load_contigs

numsites: (int) Number of sites

verbose: (bool) A value of True will force the 'flush' the buffer and everything will be written to screen.

mpa.PredictiveInfo

Overview

The predictive information class is a good way of assessing the quality of a model inferred from a massively parallel dataset.

Usage

```
>>> loader = mpathic.io
```

```
>>> dataset_df = loader.load_dataset(mpathic.__path__[0] + '/data/sortseq/
↪full-0/library.txt')
>>> mp_df = loader.load_model(mpathic.__path__[0] + '/examples/true_model.txt
↪')
    >>> ss = mpathic.SimulateSort(df=dataset_df, mp=mp_df)
>>> temp_ss = ss.output_df
```

```
>>> temp_ss = ss.output_df
>>> cols = ['ct', 'ct_0', 'ct_1', 'ct_2', 'ct_3', 'seq']
>>> temp_ss = temp_ss[cols]
>>> pi = mpathic.PredictiveInfo(data_df = temp_ss, model_df = mp_df, start=0)
>>> print(pi.out_MI)
```

Class Details

class predictive_info.**PredictiveInfo**(***kwargs*)

Parameters

data_df: (pandas data frame) Dataframe containing several columns representing bins and sequence column. The integer values in bins represent the occurrence of the sequence that bin.

model_df: (pandas dataframe) The dataframe containing a model of the binding energy and a wild type sequence

start: (int) Starting position of the sequence.

end: (int) end position of the sequence.

err: (bool) boolean variable which indiciates the inclusion of error in the mutual information estimate if true

coarse_graining_level: (int) Speed computation by coarse-graining model predictions

mpa.demo()

`mpathic.demo(example='simulation')`
Runs a demonstration of mpathic.

Parameters

example: (str) A string specifying which demo to run. Must be 'simulation', 'profile', or 'modeling'.

Returns

None.

1.4 Contact

For technical assistance or to report bugs, please contact Ammar Tareen.

For more general correspondence, please contact Justin Kinney.

Other links:

- Kinney Lab
- Simons Center for Quantitative Biology
- Cold Spring Harbor Laboratory

1.5 References

1.6 Common Installation Issues

1.6.1 Installation Issues

Fortran Compiler

1. Missing Fortran compiler

pymc requires a fortran compiler in order to work. Please ensure pymc can be imported.

```
>>> import pymc
```

During installation, MPAthic will look for existing fortran compilers on the user's machine. If none are present, the following error will be thrown:

```
get_default_fcompiler: matching types: '['gnu95', 'nag', 'absoft', 'ibm', 'intel', 'gnu', 'g95', 'pg']'
customize Gnu95FCompiler
Could not locate executable gfortran
Could not locate executable f95
customize NAGFCompiler
customize AbsoftFCompiler
Could not locate executable f90
Could not locate executable f77
customize IBMFCompiler
Could not locate executable xlf90
Could not locate executable xlf
customize IntelFCompiler
Could not locate executable ifort
Could not locate executable ifc
customize GnuFCompiler
Could not locate executable g77
customize G95FCompiler
Could not locate executable g95
customize PGroupFCompiler
Could not locate executable pgfortran
don't know how to compile Fortran code on platform 'posix'
warning: build_ext: f77_compiler=None is not available.

building 'pymc.flib' extension
error: extension 'pymc.flib' has Fortran sources but no Fortran compiler found
```

Fix

We recommend installing GCC, as this satisfies both Non-Python MPAthic dependencies (i.e. Cython and pymc). In addition to official instructions, GCC can be obtained easily on macOS via homebrew:

```
brew install gcc
```

2. Updating gcc

Updates to gcc does not seem to update the paths required by pymc. An example is shown below where the user initially installed gcc 4 but then updated to version 5:

```
>>> import pymc
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/tareen/Library/Python/2.7/lib/python/site-packages/pymc/__init__.py", line 30, in <module>
    from .CommonDeterministics import *
  File "/Users/tareen/Library/Python/2.7/lib/python/site-packages/pymc/CommonDeterministics.py", line 21, in <module>
    from .utils import safe_len, stukel_logit, stukel_invlogit, logit, invlogit, value, find_element
  File "/Users/tareen/Library/Python/2.7/lib/python/site-packages/pymc/utils.py", line 14, in <module>
    from . import flib
ImportError: dlopen(/Users/tareen/Library/Python/2.7/lib/python/site-packages/pymc/flib.so, 2): Library not loaded: /usr/local/opt/gcc/lib/gcc/7/libgfortran.4.dylib
  Referenced from: /Users/tareen/Library/Python/2.7/lib/python/site-packages/pymc/flib.so
  Reason: image not found
```

Fix

Re-installing the version of gcc required by pymc (hence, mpathic) fixes this issue. In the case above, gcc version 4 was re-installed.

Cython

Ensure the correct version of Cython is installed.

```
$ pip freeze | grep 'Cython'
$ Cython==0.28.1
```

1. Existing Cython versions

```
  Found existing installation: Cython 0.27.3
    Uninstalling Cython-0.27.3:
Exception:
Traceback (most recent call last):
  File "/anaconda2/lib/python2.7/site-packages/pip/basecommand.py", line 215, in main
    status = self.run(options, args)
  File "/anaconda2/lib/python2.7/site-packages/pip/commands/install.py", line 342, in run
    prefix=options.prefix_path,
  File "/anaconda2/lib/python2.7/site-packages/pip/req/req_set.py", line 778, in install
    requirement.uninstall(auto_confirm=True)
  File "/anaconda2/lib/python2.7/site-packages/pip/req/req_install.py", line 754, in uninstall
    paths_to_remove.remove(auto_confirm)
  File "/anaconda2/lib/python2.7/site-packages/pip/req/req_uninstall.py", line 115, in remove
    renames(path, new_path)
  File "/anaconda2/lib/python2.7/site-packages/pip/utils/__init__.py", line 267, in renames
    shutil.move(old, new)
  File "/anaconda2/lib/python2.7/shutil.py", line 316, in move
    copy2(src, real_dst)
  File "/anaconda2/lib/python2.7/shutil.py", line 144, in copy2
    copyfile(src, dst)
  File "/anaconda2/lib/python2.7/shutil.py", line 96, in copyfile
    with open(src, 'rb') as fsrc:
IOError: [Errno 2] No such file or directory: '/anaconda2/lib/python2.7/site-packages/cython'
```

2 Cython environment error

```
Found existing installation: Cython 0.26.1
  Uninstalling Cython-0.26.1:
Could not install packages due to an EnvironmentError: [Errno 2] No such file or directory:
'/Users/jkinney/anaconda3/lib/python3.6/site-packages/__pycache__/cython.cpython-36.pyc'
```

Fix

Run the anaconda command:

```
conda install -c anaconda cython
```

Or pip install directly:

```
pip install Cython==0.28.1
```

Permissions

The user might not have access to install to the global site-packages directory.

```
  File "/System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/os.py", line 157, in makedirs
    mkdir(name, mode)
OSError: [Errno 13] Permission denied: '/Library/Python/2.7/site-packages/alabaster-0.7.10.dist-info'
```

Fix

```
pip install mpathic --user
```

1.7 Indices and tables

- genindex
- modindex
- search

Index