LIMBR Documentation

Release v0.2.6-alpha

Alexander Crowell

Aug 08, 2018

Contents

1 LIMBR: Learning and Imputation for Mass-spec Bias Reduction **1**

1.1 Motivation 1

1.2 Features 1

1.3 Example Usage 2

1.4 Installation 2

1.5 API Reference 2

1.6 How to Use? 2

1.7 TO DO 5

1.8 Credits 5

1.9 Built With 6

1.10 License 6

2 LIMBR **7**

2.1 LIMBR package 7

2.1.1 Submodules 7

2.1.2 LIMBR.imputable module 7

2.1.3 LIMBR.old_fashioned module 7

2.1.4 LIMBR.sva module 8

2.1.5 Module contents 8

3 Indices and tables **9**

Python Module Index **11**

CHAPTER 1

LIMBR: Learning and Imputation for Mass-spec Bias Reduction

LIMBR provides a streamlined tool set for imputation of missing data followed by modelling and removal of batch effects. The software was designed for proteomics datasets, with an emphasis on circadian proteomics data, but can be applied to any time course or blocked experiments which produce large amounts of data, such as RNAseq. The two main classes are imputable, which performs missing data imputation, and sva, which performs modelling and removal of batch effects.

1.1 Motivation

Decreasing costs and increasing ambition are resulting in larger Mass-spec (MS) experiments. MS experiments have a few limitations which are exacerbated by this increasing scale, namely batch effects and missing data. Many downstream statistical analyses require complete cases for analysis, however, MS produces some missing data at random meaning that as the number of experiments increase the number of peptides rejected due to missing data actually *increases*. This is obviously not good, but fortunately there is a solution! If the missing data for observations missing only a small number of data points are imputed this issue can be overcome and that's the first thing that LIMBR does. The second issue with larger scale MS experiments is batch effects. As the number of samples increases, the number of batches necessary for sample processing also increases. Batch effects from sample processing are known to have a large effect on MS data and increasing the number of batches means more batch effects and a higher proportion of observations affected by at least one batch effect. Here LIMBR capitolizes on the larger amount of data and the known correlation structure of the data set to model these batch effects so that they can be removed.

1.2 Features

- KNN based imputation of missing data.
- SVA based modelling and removal of batch effects.
- Built for circadian and non-circadian time series as well as block designs

1.3 Example Usage

‘‘‘python from LIMBR import simulations, imputation, batch_fx

simulation = simulations.simulate() simulation.generate_pool_map() simulation.write_output()

#Read Raw Data to_impute = imputation.imputable(‘simulated_data_with_noise.txt’,0.3) #Impute and Write Output to_impute.impute_data(‘imputed.txt’)

#Read Imputed Data to_sva = batch_fx.sva(filename=’imputed.txt’,design=’c’,data_type=’p’,pool=’pool_map.p’) #preprocess data to_sva.preprocess_default() #perform permutation testing to_sva.perm_test(nperm=100) #write_output to_sva.output_default(‘LIMBR_processed.txt’) ‘‘‘

1.4 Installation

pip install limbr

1.5 API Reference

http://limbr.readthedocs.io/en/latest/source/modules.html

1.6 How to Use?

A Note on Data Formatting LIMBR expects input files to be formatted as tab seperated. For Proteomics data, The first column should contain the Peptide and the second column the protein to which that peptide corresponds. In the case of RNAseq data, the first column should indicate the gene or transcript identifier. The header should start with ‘Peptide’ and ‘Protein’ for proteomics data or ‘#’ for rnaseq data. For time series datasets, the rest of the header should be either of the form 02_1 for data with the first number indicating the timepoint and the second the replicate or of the form pool_01 for pooled controls. It is important that single digit timepoints include the leading zero for formatting. Missing values should bbe indicated by the string ‘NULL’. Example data file:

Peptide | Protein | 00_1 | 00_2 | 00_3 | 02_1 | 02_2 | 02_3 |

|—|—|—|—|—|—|—|—| | Peptide_ID | Protein_ID | data | data | data | data | data | data |

Before using LIMBR you need to specify a few key features of your experiment. If you are analyzing proteomics data with pooled controls, you need to let LIMBR know which pools correspond to which samples. This is done by generating a pool_map file. The pool_map is a pickled python dictionary of the form *pool_map = {‘02_1’:1}* with keys being the column headers of your samples and the values being the number of the corresponding pool. The pickled file can be generated by running *pickle.dump(pool_map, open(out_name+’.p’, “wb”))*. Similarly, if you are analyzing your data in blocked mode (i.e. a non-time course experiment) you will need to create a block file by pickling a python dictionary with keys corresponding to the column headers of your samples and values indicating the block to which that sample belongs.

Once your data is properly formatted and you’ve generated the experimental design files you need, things get much easier.

Imputing

Imputing data requires only 3 pieces of information: the path to your raw data file, the percentage of missing data beyond which you don't want to impute and the path to your desired output file. Obviously you don't want to guess values for peptides which you almost never observed, but where to draw the line? Generally imputing when <30% of values are missing is reasonable for large datasets. You probably want to impute at least all peptides for which <10% of values are missing as this is a _very_ conservative threshold and not imputing at all introduces its own biases.

- filename = PATH TO YOUR INPUT FILE
- missingness = MAXIMUM IMPUTATION LEVEL (0.3 = 30%)
- output = PATH TO YOUR DESIRED OUTPUT FILE

‘‘‘python from LIMBR import imputation

#Read Raw Data to_impute = imputation.imputable(filename,missingness) #Impute and Write Output to_impute.impute_data(output) ‘‘‘

Removing Batch Effects

Removing batch effects requires a little more information than imputing, but not much. You need to specify the path to your input file (which should be the output of imputation), the design of your experiment, whether it uses proteomic or rnaseq data, and if proteomic which pools map to which experiments. The possible experimental designs are circadian time course (‘c’), non-circadian timecourse (‘t’) or blocked (‘b’). You will also need to specify the number of permutations used to estimate the significance of bias trends. More permutations are better, but there are diminishing returns in addition to the increased time required. In simulated datasets, LIMBR performs very well with even 100 permutations, however 10,000 permutations can be performed on even very large datasets in around 2 hours.

- filename = PATH TO YOUR INPUT FILE (output of imputation)
- design = EXPERIMENTAL DESIGN (‘c’ = circadian time course, ‘t’ = non-circadian time course, ‘b’ = blocked)
- data_type = DATA TYPE (‘p’ = proteomic, ‘r’ = rnaseq)
- pool = PATH TO POOL MAP FILE
- nperm = NUMBER OF PERMUTATIONS
- output = PATH TO DESIRED OUTPUT FILE

‘‘‘python from LIMBR import batch_fx

#Read Imputed Data (‘c’ indicates circadian experimental design, ‘p’ indicates proteomic data type) to_sva = batch_fx.sva(filename,design,data_type,pool) #preprocess data to_sva.preprocess_default() #perform permutation testing to_sva.perm_test(nperm) #write_output to_sva.output_default(output) ‘‘‘

And that's it, your data is ready for downstream analysis!

More Control

If you need more control, you can skip the helper functions shown above and run LIMBR step by step, supplying alternatives to the default parameters where desired.

- filename = PATH TO YOUR INPUT FILE (output of imputation)
- design = EXPERIMENTAL DESIGN (‘c’ = circadian time course, ‘t’ = non-circadian time course, ‘b’ = blocked)
- data_type = DATA TYPE (‘p’ = proteomic, ‘r’ = rnaseq)
- pool = PATH TO POOL MAP FILE
- perc_red = PERCENTAGE BY WHICH TO REDUCE DATA (25 = 25%)
- nperm = NUMBER OF PERMUTATIONS

- npr = NUMBER OF PROCESSORS (for permutation testing, incompatible with MAC - leave set to 1)
- alpha = SIGNIFICANCE CUTOFF FOR BATCH EFFECTS
- lam = BACKGROUND CUTOFF (for estimation of association between peptides and batch effects)
- output = PATH TO DESIRED OUTPUT FILE

‘‘‘ from LIMBR import batch_fx

#import data to_sva = batch_fx.sva(filename,design,data_type,pool) #normalize for pooled controls to_sva.pool_normalize() #calculate timepoints from header to_sva.get_tpoints() #calculate correlation with primary trend of interest to_sva.prim_cor() #reduce data based on primary trend correlation to_sva.reduce(perc_red) #calculate residuals to_sva.set_res() #calculate tks to_sva.set_tks() #perform permutation testing to_sva.perm_test(nperm,npr) #perform eigen trend regression to_sva.eig_reg(alpha) #perform subset svd to_sva.subset_svd(lam) #write_output to_sva.normalize(output) ‘‘‘

Performance

So how does LIMBR do on that simulated data from the first useage example? One simple way to test would be to run LIMBRs output through eJTK along with the output of a simpler normalization procedure and compare the ROC curves. eJTK is an algorithm for classification of circadian expression by Alan Hutchison which can be found [here](https://github.com/alanlhutchison/empirical-JTK_CYCLE-with-asymmetry). To get the output of a basic normalization protocol we can do:

‘‘‘python from LIMBR import old_fashioned

to_old = old_fashioned.old_fashioned(filename=’simulated_data_with_noise.txt’,data_type=’p’,pool=’pool_map.p’) to_old.pool_normalize() to_old.normalize(‘old_processed.txt’) ‘‘‘

When you generated your simulated data, the simulation module should also have output a ‘baseline’ data file. This file contains the simulated data before the addition of any bias trends, which we can use to set a performance baseline (we would never expect an algorithm to perform better than the results we get from analyzing the baseline data).

If you have eJTK installed in a ./src directory relative to the location of the files generated by LIMBR, from bash, you can then run:

‘‘‘bash sed -e ‘s/_[[:digit:]]//g’ LIMBR_processed.txt > temp.txt cut -f 1,3- temp.txt > LIMBR_processed.txt sed -e ‘s/_[[:digit:]]//g’ old_processed.txt > temp.txt cut -f 1,3- temp.txt > old_processed.txt sed -e ‘s/_[[:digit:]]//g’ simulated_data_baseline.txt > temp.txt cut -f 1,3- temp.txt > simulated_data_baseline.txt rm temp.txt

python2 src/eJTK-CalcP.py -f LIMBR_processed.txt -w src/ref_files/waveform_cosine.txt -a src/ref_files/asymmetries_02-22_by2.txt -s src/ref_files/phases_00-22_by2.txt -p src/ref_files/period24.txt

python2 src/eJTK-CalcP.py -f old_processed.txt -w src/ref_files/waveform_cosine.txt -a src/ref_files/asymmetries_02-22_by2.txt -s src/ref_files/phases_00-22_by2.txt -p src/ref_files/period24.txt

python2 src/eJTK-CalcP.py -f simulated_data_baseline.txt -w src/ref_files/waveform_cosine.txt -a src/ref_files/asymmetries_02-22_by2.txt -s src/ref_files/phases_00-22_by2.txt -p src/ref_files/period24.txt ‘‘‘

The first part simply removes the unique replicate identifiers from the headers of our files to comply with eJTKs formatting conventions and the second part runs eJTK. This should result in several output files including *LIMBR_processed__jtkout_GammaP.txt* and *old_processed__jtkout_GammaP.txt*. The ‘true_classes’ file used here should have been generated when you ran the simulation module. Once this classification step is complete, LIMBR can help you analyze your results. Back in python you can run:

‘‘‘python from LIMBR import simulations

analysis = simulations.analyze(‘simulated_data_true_classes.txt’) analysis.add_data(‘LIMBR_processed__jtkout_GammaP.txt’,’LIMBR
analysis.add_data(‘old_processed__jtkout_GammaP.txt’,’traditional’) analysis.add_data(‘simulated_data_baseline__jtkout_GammaP.txt
analysis.generate_roc_curve() ‘‘‘

You should get a ROC curve that looks something like this:

![ImageRelative](images/ROC_1000.png “ROC_1000”)

LIMBR should clearly outperform the traditional method, but not quite reach the level of the baseline. It’s important to remember that LIMBR works better with larger datasets from which to learn. If we repeat the above example with all the same parameters but increase the number of rows of data to 10,000, we get ROC curves which look like this:

![ImageRelative](images/ROC_10000.png “ROC_10000”)

While this example takes longer to run, the performance is clearly superior. 10,000 rows is still relatively small for biological data, so it’s reasonable to expect higher performance and longer run times in practice than what you see in the examples.

Further Exploration

If you’d like to further explore LIMBR, there are several additional parameters that can be tweaked in generating simulated datasets.

- points = NUMBER OF TIMEPOINTS
- nrows = NUMBER OF ROWS OF DATA
- nreps = NUMBER OF REPLICATES
- tpoint_space = AMOUNT OF TIME BETWEEN TIMEPOINTS
- pcirc = PROBABILITY OF A PEPTIDE BEING CIRCADIAN
- phase_prop = PROPORTION OF PEPTIDES IN EACH PHASE GROUP (two phases of expression)
- phase_noise = AMOUNT OF VARIABILITY IN PHASE WITHIN PHASE GROUPS
- amp_noise = AMOUNT OF BIOLOGICAL VARIABILITY IN EXPRESSION
- n_batch_effects = NUMBER OF BATCH EFFECTS
- pbatch = PROBABILITY OF A PEPTIDE BEING EFFECTED BY EACH BATCH EFFECT
- effect_size = AVERAGE MAGNITUDE OF BATCH EFFECTS
- p_miss = PROBABILITY OF A PEPTIDE MISSING ANY DATA
- lam_miss = POISSON LAMBDA FOR HOW MANY OBSERVATIONS MISSING IF ANY

```
` simulation = simulations.simulate(tpoints, nrows, nreps, tpoint_space,
pcirc, phase_prop, phase_noise, amp_noise, n_batch_effects, pbatch,
effect_size, p_miss, lam_miss) `
```

1.7 TO DO

- Implement simulations for non-circadian time courses and block designs.
- Add unit tests to docstrings where possible.
- Review ensuring maximum Vectorization/CUDA implementation.
- Improve eJTK integration.

1.8 Credits

K nearest neighbors as an imputation method was originally proposed by Gustavo Batista in 2002 (http://conteudo.icmc.usp.br/pessoas/gbatista/files/his2002.pdf) and has seen a great deal of success since.

The sva based methods build on work for micro-array datasets by Jeffrey Leek, with particular reliance on his PhD Thesis from the University of Washington (https://digital.lib.washington.edu/researchworks/bitstream/handle/1773/9586/3290558.pdf?sequence=1).

1.9 Built With

- numpy
- pandas
- scipy
- sklearn
- statsmodels
- tqdm
- multiprocess
- matplotlib

1.10 License

CHAPTER 2

LIMBR

2.1 LIMBR package

2.1.1 Submodules

2.1.2 LIMBR.imputable module

2.1.3 LIMBR.old_fashioned module

class LIMBR.old_fashioned.**old_fashioned**(*filename*, *data_type*, *pool=None*)

Bases: `object`

Performs a standard normalization procedure without SVD as a baseline.

This class performs simple quantile normalization and row scaling along with pool normalization for proteomics experiments using the same methods and interface employed in the sva class. This provides a baseline comparison point for data processed with LIMBR.

Parameters

- **filename** (*str*) – Path to the input dataset.
- **data_type** (*str*) – Type of dataset, one of 'p' or 'r'. 'p' indicates proteomic with two index columns specifying peptide and protein. 'r' indicates RNAseq with one index column indicating gene.
- **pool** (*str*) – Path to file containing pooled control design for experiment in the case of data_type = 'p'. This should be a pickled dictionary with the keys being column headers corresponding to each sample and the values being the corresponding pooled control number.

Variables

- **raw_data** (*dataframe*) – This is where the input data is stored.
- **data_type** (*str*) – This is where the data type ('p' or 'r') is stored.

- **norm_map** (*dict*) – This is where the assignment of pooled controls to samples are stored if data_type = 'p'.

normalize(*outname*)

Groups peptides by protein and outputs final processed dataset.

These final results are then written to an output file.

Parameters **outname** (*str*) – Path to desired output file.

pool_normalize()

Preprocessing normalization.

Performs pool normalization on an sva object using the raw_data and norm_map if pooled controls were used. Quantile normalization of each column and scaling of each row are then performed.

Variables

- **scaler** (*sklearn.preprocessing.StandardScaler()*) – A fitted scaler from the sklearn preprocessing module.
- **data_pnorm** (*dataframe*) – Pool normalized data.

2.1.4 LIMBR.sva module

2.1.5 Module contents

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

l

LIMBR, 8
LIMBR.old_fashioned, 7

Index

L

LIMBR (module), 8

LIMBR.old_fashioned (module), 7

N

normalize() (LIMBR.old_fashioned.old_fashioned method), 8

O

old_fashioned (class in LIMBR.old_fashioned), 7

P

pool_normalize() (LIMBR.old_fashioned.old_fashioned method), 8