rambutan Documentation

Release 0.1

Jacob Schreiber

Aug 07, 2018

Contents

1 Installation **3**

2 Frequently Asked Questions **5**

3 Rambutan **7**

4 API Reference **9**

5 Data Generators **13**
5.1 API Reference 13

6 Utilities **15**
6.1 API Reference 15

Python Module Index **19**

Rambutan is a package for the prediction of the 3D structure of human cell types. It focuses on the prediction of Hi-C contact maps, but rather than trying to predict the number of contacts that a pair of loci engage in, it instead predicts whether the contact is statistically significant with respect to their genomic distance. This genomic distance effect is extremely important as pairs of loci that are close together are very likely to be in contact simply due to physics as opposed to biological importance, whereas long-range contacts are typically enriched for important biological interactions. The predictions are made using a convolutional neural network that takes in nucleotide sequence and DNaseI sensitivity from two loci spanning 1000 nucleotides. The goal is to serve as a substitute for both running an experiment to collect Hi-C data for a cell type and for running Fit-Hi-C on the data afterwards.

Rambutan solves a big data problem. This means that it is not optimized for use on laptops or old computers, but rather works best on modern computational servers. Most experiments were done using a computer with four K40 GPUs and over 60GB of RAM. To illustrate this, there are over 30 million pairs of positions in chromosome 21, the smallest chromosome, within the band considered by Rambutan. Using all four GPUs, it can easily take approximately an hour for this prediction. While it's certainly possible to run Rambutan even on only the CPU, one must make up for with patience what they lack in hardware.

The manuscript is currently under revisions at Bioinformatics, but the preprint on bioRxiv can be found here: http://www.biorxiv.org/content/early/2017/01/30/103614

Comments and suggestions are always greatly appreciated.

CHAPTER 1

Installation

Installation of Rambutan is complicated by two of its dependencies– mxnet and cython. If you have these two packages installed, then all you need to do is download the current code with the following commands:

```
git clone https://github.com/jmschrei/rambutan
cd rambutan
python setup.py install
```

It is difficult to install either mxnet or cython on a windows machine. mxnet has a great deal of dependencies that can dramatically influence the speed of Rambutan and should not be ignored. In particular, cuDNN is very valuable when it comes to accelerating the models.

mxnet installation instructions can be found at http://mxnet.io/get_started/install.html

cython requires a working c++ compiler. Both Mac (clang) and Linux (gcc) machines come with working compilers. For Python 2, this minimal version of Visual Studio 2008 has worked well in the past: https://www.microsoft.com/en-us/download/details.aspx?id=44266. For Python 3, this version of the Visual Studio Build Tools has worked in the past: http://go.microsoft.com/fwlink/?LinkId=691126. If neither of those work, then https://wiki.python.org/moin/WindowsCompilers may provide more information. Note that your compiler version must match your python version. Run *python –version* to tell which python version you use. Don't forget to select the appropriate Windows version API you'd like to use. If you get an error message "ValueError: Unknown MS Compiler version 1900" remove your Python's Lib/distutils/distutil.cfg and retry. See http://stackoverflow.com/questions/34135280/valueerror-unknown-ms-compiler-version-1900 for details.

It is suggested that one start out with the Anaconda or Enthought python distributions that come with all other dependencies pre-installed. This will significant speed up the starting time for those who are getting started with Python.

CHAPTER 2

Frequently Asked Questions

17. What exactly is Rambutan predicting?

1. Rambutan is predicting the probability that two regions in the genome engage in a statistically significant contact, given their nucleotide sequences and DNaseI sensitivity data. Statistical significance is denoted as a q-value (FDR) of 0.01 or less according to Fit-Hi-C.

17. Most contact matrices are made up of integer counts, why aren't you predicting this?

1. It is true that the output data from a Hi-C matrix comes in the form of an integer number of counts that a pair of regions engage, as determined by deep sequencing runs. However, this number is highly dependent on the sequencing depth of the experiment, which can change from one experiment to the next. In addition, these values suffer from several biases, including a region-specific bias that is influenced by, among other things, GC content, and the very powerful genomic distance effect. Ultimately when considering 3D genome structure one does not think about how many times a pair of regions are in contact according to a sequencing experiment, but simply the binary of whether they are in contact or not. Rambutan's predictions accounts for the biases, and predicts this, arguably more useful measurement of genomic structure.

17. Can I train Rambutan using a different epigenetic mark?

1. Absolutely. The toolset is already there, since you can feed in an arbitrary bit-encoded epigenetic data in the place of the DNaseI values. The only restriction is that the input must have 8 bits at each position. If you need fewer than 8 bits to encode your epigenetic data it is perfectly acceptable to leave of the bits always off.

17. Can I use bedgraph_to_dense and encode_dnase to bit encode my different epigenetic mark?

1. Yes, as long as the range of the values is between -2 and 5, or you are fine with limiting the dynamic range of the signal to that width. If you have some other file that falls within this range that you'd like to use instead, you can treat that signal identically to how you would treat DNaseI.

17. Why did you restrict the input to only DNaseI signal, won't additional epigenetic marks improve accuracy?

1. It is likely the case that more epigenetic marks will improve accuracy. However, the point of the model is not solely to produce accurate models but to serve as a replacement for running high resolution Hi-C experiments. If a researcher has to run several epigenetic experiments for a cell type that doesn't currently have all of the marks before being able to run Rambutan, it may be worth it for them to simply run the Hi-C experiment. DNaseI signal has been collected for over 400 human cell types, and is cheap and easy to gather for future ones.

17. How powerful a machine is necessary?

1. The problem is a big data problem so it is unlikely that it can be solved in a reasonable amount of time on a laptop. A user should have at least one GPU and at least 16G of RAM.

17. Are GPUs necessary to make predictions?

1. Technically it is possible to use only CPUs to make predictions but it is unlikely that this will

17. Why is the project named Rambutan?

1. This question is left as an exercise for the user.

CHAPTER 3

Rambutan

This file contains the Rambutan model. It is composed of two functions, *fit* and *predict*.

The *predict* method will use a pre-trained version of Rambutan and predict all pairs of regions in a given sequence file. It takes in nucleotide sequence and DNaseI sensitivity either as bit encoded numpy arrays, or as FastA and bedgraph files respectively. Note that the DNaseI sensitivity should be the fold change signal, not the log10 p-values or raw values. If FastA or bedgraph files are passed in, they are converted internally to the bit encoding. Here is an example of it's usage:

```
from rambutan import Rambutan

model = Rambutan('rambutan', 25, verbose=True)
y_pred = model.predict('chr21.fa', 'E003-DNase.chr21.fc.signal.bedgraph', ctxs=[0, 1,␣
↪2, 3])
```

This assumes that the directory that you are running the code has the Rambutan model and parameter files (for the 25th iterator), the FastA file for chr21, and the appropriate bedgraph file. It will also use gpus 0, 1, 2, and 3 to make the predictions. If you have fewer gpus, pass in fewer numbers.

The *fit* function will accept training (and optionally validation) data either as FastA/bedgraph/HiC files, or as numpy array/numpy array/HiC files. It will automatically convert the FastA and bedgraph files to the bit encoding internally. However, this main require a great deal of memory, so it is recommended that you preprocess your data to numpy arrays, store them to disk, and then feed them in as memory maps so that the entire genome does not need to be stored in RAM for Rambutan to be trained.

Here is an example of how to pre-encode your data:

```
from rambutan.utils import fasta_to_dense
from rambutan.utils import bedgraph_to_dense
from rambutan.utils import encode_dnase

# Process the sequence files first
sequence_files = ['chr{}.fa'.format(i) for i in range(1, 23)]
for i, sequence_file in enumerate(sequence_files):
        sequence = fasta_to_dense(sequence_file, verbose=True)
```

(continues on next page)

(continued from previous page)

```
        numpy.save('chr{}.ohe.npy'.format(i), sequence)

# Process the DNaseI files next
dnase_files = ['E003-DNase.chr{}.fc.signal.bedgraph'.format(i) for i in range(1, 23)]
for i, dnase_file in enumerate(dnase_files):
        dnase = bedgraph_to_dense(dnase_file, verbose=True)
        dnase_encoded = encode_dnase(dnase, verbose=True)
        numpy.save('chr{}.dnase.npy'.format(i), dnase_encoded)
```

After running this you will now have the nucleotide sequence and DNaseI inputs bit encoded as used in the Rambutan model. You can now load them up as memory maps, which essentially are pointers to where the data lives on disk. This means that the entire array doesn't get loaded into memory, but only subsections are loaded as needed. This will slow down training time, but can be mitigated if a solid state drive is used. Here is an example of fitting using memory maps:

```
train_contacts = 'GM12878_combined.res1000.contacts.txt.gz'
train_sequence = numpy.array([ numpy.load('chr{}.ohe.npy'.format(i), mmap_mode='r')␣
↪for i in chromosomes ])
train_dnases   = numpy.array([ numpy.load('chr{}.dnase.npy'.format(i), mmap_mode='r')␣
↪for i in chromosomes ])

model = Rambutan()
model.fit(train_sequence, train_dnase, train_contacts, ctxs=[0, 1, 2, 3])
```

This will fit the Rambutan model to the given data using four GPUs. Fewer or more GPUs can be used as long as they are in the same machine. Distributed learning is not yet available for training Rambutan models.

CHAPTER 4

API Reference

class rambutan.rambutan.**Rambutan**(*name='rambutan'*, *iteration=None*, *model=None*, *learning_rate=0.01*, *num_epoch=25*, *epoch_size=500*, *wd=0.0*, *optimizer='adam'*, *batch_size=1024*, *min_dist=50000*, *max_dist=1000000*, *use_seq=True*, *use_dnase=True*, *use_dist=True*, *verbose=True*)

Rambutan: a predictor of mid-range DNA-DNA contacts.

This serves as a wrapper for all functionality involving the use of Rambutan. There are two main functions to use, fit and predict. Fit involves taking in nucleotide sequence, DNaseI sensitivity, and a contact map, and training the model. Predict involves taking in nucleotide sequence and DNaseI sensitivity and predicting significant contacts.

Note: Due to a limitation of mxnets part, you cannot fit and predict in the same program. You must fit the model and save the parameters during training, and then load the pre-fit model and make predictions.

Parameters

name [str, optional] The name of the model, necessary for saving or loading parameters. Default is 'rambutan'.

iteration [int or None, optional] The iteration of training to load model parameters from, if using Rambutan in predict mode. Default is None.

model [mxnet.symbol or None] An alternate neural network can be passed in if one wishes to train that using the same framework instead of the original Rambutan model.

learning_rate [float, optional] The learning rate for the optimizer. Default is 0.01.

num_epoch [int, optional] The number of epochs to train the model for. Default is 25.

epoch_size [int, optional] The number of batches which comprise an 'epoch'. Default is 500.

wd [float, optional] The weight decay. This is equivalent to L2 regularization. Default is 0.0.

optimizer [str, optional] The optimizer to use for training. Default is 'adam'.

batch_size [int, optional] The number of samples to use in each batch. Default is 1024.

min_dist [int, optional] The minimum distance to consider contacts for. Default is 50kb.

max_dist [int, optional] The maximum distance to consider contacts for. Default is 1mb.

use_seq [bool, optional] Whether to use nucleotide sequence as an input to the model in the training step. Default is True.

use_dnase [bool, optional] Whether to use DNaseI sensitivity as an input to the model in the training step. Default is True.

use_dist [bool, optional] Whether to use genomic distance as an input to the model in the training step. Default is True.

verbose [bool, optional] Whether to output information during training and prediction. Default is True.

fit (*sequence, dnase, contacts, regions=None, validation_contacts=None, training_chromosome=None, validation_chromosome=None, ctxs=[0], eval_metric='acc', symbol=None, n_jobs=1*)
Fit the model to sequence, DNaseI, and Hi-C data.

You can fit the Rambutan model to new data. One must pass in sequence data, DNaseI data, and Hi-C contact maps. The sequence data can come either in the form of FastA files or one-hot encoded numpy arrays. The DNaseI data can likewise come as either bedgraph files or numpy arrays. The Hi-C data must come in the traditional 7 column format. Validation data can optionally be passed in to report a validation set error during the training process. NOTE: Regardless of if they are used or not, all chromosomes should be passed in to the *sequence* and *dnase* parameters. The contacts specified in *contacts* will dictate which are used. This is to make the internals easier.

Parameters for training such as the number of epochs and batches are set in the initial constructor, following with the sklearn format for estimators.

Parameters

sequence [numpy.ndarray, shape (n, 4) or str] The nucleotide sequence. Either a one hot encoded matrix of nucleotides with n being the size of the chromosome, or a file name for a fasta file.

dnase [numpy.ndarray, shape (n, 8) or str] The DNaseI fold change sensitivity. Either an encoded matrix in the manner described in the manuscript or the file name of a bedgraph file.

regions [numpy.ndarray or None, optional] The regions of interest to look at. All other regions will be ignored. If set to none, the regions of interest are defined to be 1kb bins for which all nucleotides are mappable, i.e. where there are no n or N symbols in the fasta file. Default is None.

ctxs: list, optional The contexts of the gpus to use for prediction. Currently prediction is only supported on gpus and not cpus due to the time it would take for prediction. For example, if you wanted to use three gpus of index 0 1 and 3 (because 2 is busy doing something else) you would just pass in ctxs=[0, 1, 3] and the prediction task will be naturally parallelized across your 3 gpus with a linear speedup.

Returns

y [numpy.ndarray, shape=(m, m)] A matrix of predictions of shape (m, m) where m is the number of 1kb loci in the chromosome. The predictions will reside in the upper triangle of the matrix since predictions are symmetric.

predict (*sequence, dnase, regions=None, ctxs=[0], sparse=False*)
Make predictions and return the matrix of probabilities.

Rambutan will make a prediction for each pair of genomic loci defined in 'regions' which fall between 'min_dist' and 'max_dist'. Inputs can either be appropriately encoded sequence and dnase files, or fasta

files and bedgraph files for the nucleotide sequence and DNaseI sensitivity respectively. Note: fasta files and bedgraph files must be made up of a single chromosome, not one entry per chromosome.

Parameters

sequence [numpy.ndarray, shape (n, 4) or str] The nucleotide sequence. Either a one hot encoded matrix of nucleotides with n being the size of the chromosome, or a file name for a fasta file.

dnase [numpy.ndarray, shape (n, 8) or str] The DNaseI fold change sensitivity. Either an encoded matrix in the manner described in the manuscript or the file name of a bedgraph file.

regions [numpy.ndarray or None, optional] The regions of interest to look at. All other regions will be ignored. If set to none, the regions of interest are defined to be 1kb bins for which all nucleotides are mappable, i.e. where there are no n or N symbols in the fasta file. Default is None.

ctxs [list, optional] The contexts of the gpus to use for prediction. Currently prediction is only supported on gpus and not cpus due to the time it would take for prediction. For example, if you wanted to use three gpus of index 0 1 and 3 (because 2 is busy doing something else) you would just pass in ctxs=[0, 1, 3] and the prediction task will be naturally parallelized across your 3 gpus with a linear speedup.

sparse [bool, optional] Whether to return three arrays, the rows, columns, and values, or the full dense matrix. Sparse is useful for large matrices.

Returns

y [numpy.ndarray, shape=(m, m)] A matrix of predictions of shape (m, m) where m is the number of 1kb loci in the chromosome. The predictions will reside in the upper triangle of the matrix since predictions are symmetric.

rambutan.rambutan.**extract_dnase**(*filename*, *verbose=False*)

Extract a DNaseI file and encode it.

This function will read in a bedgraph format file and convert it to the one-hot encoded numpy array used internally. If a one-hot encoded file is passed in, it is simple returned. This function is a convenient wrapper for joblib to parallelize the unzipping portion.

Parameters

filename [str or numpy.ndarray] The name of the bedgraph file to open or the one-hot encoded sequence.

verbose: bool, optional Whether to report the status while extracting sequence. This does not look good when done in parallel, so it is suggested it is set to false in that case.

Returns

sequence [numpy.ndarray, shape=(n, 8)] The one-hot encoded DNaseI sequence.

rambutan.rambutan.**extract_sequence**(*filename*, *verbose=False*)

Extract a nucleotide sequence from a file and encode it.

This function will read in a FastA formatted DNA file and convert it to be a one-hot encoded numpy array for internal use. If a one-hot encoded file is passed in, it is simply returned. This function is a convenient wrapper for joblib to parallelize the unzipping portion.

Parameters

filename [str or numpy.ndarray] The name of the fasta file to open or the one-hot encoded sequence.

verbose: bool, optional Whether to report the status while extracting sequence. This does not look good when done in parallel, so it is suggested it is set to false in that case.

Returns

sequence [numpy.ndarray, shape=(n, 4)] The one-hot encoded DNA sequence.

CHAPTER 5

Data Generators

Rambutan uses two data generators, the training generator and the validation generator. Both take in regions of the genome, both one-hot encoded nucleotide sequence and bit encoded DNaseI sequence, and output a random sample of pairs of regions for the Rambutan model. Essentially, minibatches are created on the fly from 1D genome data because the nucleotide level input for all pairs in the genome cannot possibly fit in memory. The major difference between the two is that the training generator randomly produces minibatches over all chromosomes that it is fed, whereas the validation generator will systematically yield all positive samples once with an equal number of negative samples. This allows an entire chromosome to be used as a validation set while not double counting regions.

5.1 API Reference

The data generators are stored here. These generators produce the examples used for training a Rambutan model.

class rambutan.io.**TrainingGenerator**

Generator iterator, collects batches from a generator.

Parameters

data [generator]

batch_size [int] Batch Size

last_batch_handle ['pad', 'discard' or 'roll_over'] How to handle the last batch

provide_data

The name and shape of data provided by this iterator

provide_label

The name and shape of label provided by this iterator

class rambutan.io.**ValidationGenerator**

Generator iterator, collects batches from a generator showing a full subset.

Use on only one chromosome for now.

provide_data
The name and shape of data provided by this iterator

provide_label
The name and shape of label provided by this iterator

CHAPTER 6

Utilities

These functions have been implemented to help users use the core Rambutan model. They consist mostly of data converters. For example, one can convert a FastA file to a one-hot encoded numpy array using the fasta_to_dense function, or calculation the insulation score of a predicted or real contact map using the insulation_score function.

6.1 API Reference

This file defines useful utility functions.

rambutan.utils.**bedgraph_to_dense**()
Read a bedgraph file and return a dense numpy array.

This will read in an arbitrary bedgraph file and turn it into a dense array for faster indexing in the future.

Parameters

filename [str] The name of the bedgraph file to use.

Returns

array [numpy.ndarray, shape=(n,)] A dense array of the unpacked values.

rambutan.utils.**benjamini_hochberg**()
Run the benjamini hochberg procedure on a vector of -sorted- p-values.

Runs the procedure on a vector of p-values, and returns the q-values for each point.

Parameters

p_values [numpy.ndarray] A vector of p values

n [int] The number of tests which have been run.

Returns

q_values [numpy.ndarray] The q-values for each point.

rambutan.utils.**count_band_regions**()

Calculate the number of regions in the band.

This will iterate over all region pairs and identify the number of region pairs within the given band. This corresponds to the number of tests.

Parameters

regions [numpy.ndarray] The mappable regions in a chromosome.

min_distance [int, optional] The minimum distance that a contact must be within.

max_distance [int, optional] The maximum distance that a contact must be within.

Returns

n [int] The number of region pairs in the band.

rambutan.utils.**downsample**()

Downsample a 1kb resolution matrix to a 5kb resolution matrix.

For each cell in the 5kb resolution matrix, take the maximum probability for each cell in the 5x5 grid at the 1kb resolution centered at this point. For example, the cell in the 5kb resolution matrix at 2500,2500 will take the maximum probability of the cells at 500,500, 500,1500, 500,2500... 4500,500, 4500,1500... etc. This is equivalent to treating the cells as being strongly correlated instead of independent from each other.

Parameters

x [numpy.ndarray, shape=(n, n)] The 1kb resolution matrix to downsample

regions [numpy.ndarray, shape=(m,)] The relevant regions to look at

min_dist [int, optional] The minimum distance two regions have to be from each other to be considered. Default is 50kb.

max_dist [int, optional] The maximum distance two regions can be from each other to be considered. Default is 1Mb.

Returns

y [numpy.ndarray, shape=(n/5, n/5)] The 5kb resolution matrix produced from the 1kb matrix.

rambutan.utils.**encode_dnase**()

Take in an array of real DNase values and binary encode the log.

This transforms the fold change value to the log fold value and then encodes this value as a binarization of the rounded log value. This is done to balance variance between enrichments and depletions.

For example, a log fold enrichment of 2.9 would have bits 0 1 2and 3 active, whereas a value of -2.2 would have bits 0, -1, and -2 active. This encodes DNase values between -2 and 5, so 8 total bits for each position.

Parameters

dnase [numpy.ndarray, shape=(n,)] The dnase fold change values read from a bedgraph or bigwig file, ranging from near 0 to above.

Returns

encoded_dnase [numpy.ndarray, shape=(n, 8)] The encoded log fold change values.

rambutan.utils.**extract_contacts**()

Extract the statistically significant contacts

Extract all contacts that have a p-value <= alpha and are within a certain band in the chromosome. This is useful for drastically reducing the size of the dataset. The columns must be named as follows:

chr1 fragmentMid1 chr2 fragmentMid2 p-value q-value

Parameters

contacts [pandas.DataFrame] The data formatted properly in a dataframe.

alpha [double, optional] The p-value threshold to filter by. Default is 0.01.

min_distance [int, optional] The minimum distance that a contact must be within.

max_distance [int, optional] The maximum distance that a contact must be within.

Returns

contacts [pandas.DataFrame] The filtered contacts within a certain threshold.

n_region_pairs [int] The number of region pairs in the band

rambutan.utils.**extract_regions**()

Extract the mappable regions for predcitions.

The mappable regions in this case are defined by those regions which have no unmappable ('N') nucleotides in the FASTA file.

Parameters

sequence [numpy.ndarray, shape=(n, 4)] The one hot encoded sequence numpy array.

Returns

regions [numpy.ndarray, shape=(m,)] The set of mappable regions (midpoints) from this file.

rambutan.utils.**fasta_to_dense**()

Translate the sequence from a file to a one hot encoded dense array.

Parameters

filename [str] The name of the fasta file to use.

Returns

array [numpy.ndarray, shape=(n, 4)] A dense array one-hot encoded for a nucleotide. 'N' does not take a value.

rambutan.utils.**insulation_score**()

Calculate the insulation score for a given matrix of any resolution.

This will slide a size*size square along the diagonal of the matrix, summing the values in the upper triangle of the matrix. If a region has no contacts it will not have an insulation score, which handles both edge cases.

Parameters

x [numpy.ndarray, shape=(n, n)] The matrix to calculate the insulation score on. Either Rambutan predictions or contacts from a Hi-C map.

size [int, optional] The size of the square, default is 200, which is 1Mb on a 5kb resolution map.

Returns

insulation [numpy.ndarray, shape=(n,)]

Python Module Index

r

rambutan.io, 13
rambutan.rambutan, 9
rambutan.utils, 15

Index

B

bedgraph_to_dense() (in module rambutan.utils), 15
benjamini_hochberg() (in module rambutan.utils), 15

C

count_band_regions() (in module rambutan.utils), 15

D

downsample() (in module rambutan.utils), 16

E

encode_dnase() (in module rambutan.utils), 16
extract_contacts() (in module rambutan.utils), 16
extract_dnase() (in module rambutan.rambutan), 11
extract_regions() (in module rambutan.utils), 17
extract_sequence() (in module rambutan.rambutan), 11

F

fasta_to_dense() (in module rambutan.utils), 17
fit() (rambutan.rambutan.Rambutan method), 10

I

insulation_score() (in module rambutan.utils), 17

P

predict() (rambutan.rambutan.Rambutan method), 10
provide_data (rambutan.io.TrainingGenerator attribute), 13
provide_data (rambutan.io.ValidationGenerator attribute), 13
provide_label (rambutan.io.TrainingGenerator attribute), 13
provide_label (rambutan.io.ValidationGenerator attribute), 14

R

Rambutan (class in rambutan.rambutan), 9
rambutan.io (module), 13
rambutan.rambutan (module), 9
rambutan.utils (module), 15

T

TrainingGenerator (class in rambutan.io), 13

V

ValidationGenerator (class in rambutan.io), 13