BEAR Documentation

Contributors

May 26, 2021

CONTENTS

CHAPTER

ONE

INSTALLATION

To install the python package, use a clean Python 3 environment and run:

```
pip install --upgrade pip
pip install .
```

Preprocessing fasta and fastq files requires installation of KMC. This code has been tested on KMC version 3.1.1. Mac and Linux users should run `ulimit -n 2048` to allow KMC to make a large number of files; otherwise the preprocessing stage may fail with a *File not found* error. The folder with the KMC binaries should be added to the path; for instance, run `export PATH="LOCATION/KMC3.1.1:$PATH"` on Mac/Linux.

To test your installation, run `pytest` in the project directory. All tests should pass.

CHAPTER

TWO

INTRODUCTION

This is a library for building Bayesian embedded autoregressive (BEAR) models, proposed in A generative nonparametric Bayesian model for whole genomes. It provides (1) a script for extracting summary statistics from large sequence datasets (which relies on KMC), (2) a python package with tools for building BEAR models in general (which relies on TensorFlow), and (3) scripts for training and evaluating specific example BEAR models. To use the package, follow the *installation instructions* and clone the **`package repository`_**. To get started with an example dataset, jump to the *tutorial*.

CHAPTER

THREE

PREPROCESSING SEQUENCE DATA

3.1 summarize.py

Extract summary statistics (kmer count transitions) from large nucleotide datasets in order to train BEAR models. Usage:

```
python summarize.py file out_prefix [-l L] [-mk MK] [-mf MF] [-p P] [-t T]
```

3.1.1 Input

file [str] The input csv file with rows in the format *FILE, GROUP, TYPE* where *FILE* is a path, *GROUP* is an integer, and *TYPE* is either *fa* (denoting fasta) or *fq* (denoting fastq). Files with the same group will have their counts merged. All files must contain DNA sequences (A, C, G, and T alone).

out_prefix [str] The file name prefix (including path) for the output files.

-l [int, default = 10] The maximum lag (the truncation level).

-mk [float, default = 12] Maximum amount of memory available, in gigabytes (corresponding to the KMC -m flag).

-mf [float, default = 0.1] Maximum size of output files, in gigabytes.

-p [str, default = ‘’] Path to KMC binaries. If these binaries are in your PATH, there is no need to use this option.

-t [str, default = ‘tests/exdata/tmp/’] Folder in which to store KMC’s intermediate results. A valid path MUST be provided.

You can run `python summarize.py -h` for help and more advanced options.

Output:

A collection of files:

```
out_prefix_lag_1_0.tsv, ..., out_prefix_lag_1_N.tsv, ...,
out_prefix_lag_L_0.tsv, ..., out_prefix_lag_L_N.tsv
```

where L is the maximum lag and there are N total files for each lag. Each file is a tsv with rows of the format:

```
kmer        [[transition counts in group 0 files],[transition counts in group 1 files],..
↪.]
```

The symbol *[* in the kmer is the start symbol. Each counts vector is in the order *A, C, G, T, $* where $ is the stop symbol.

Caution: KMC discards kmers with more than 4 billion counts, which may lead to errors on ultra large scale datasets.

Caution: The script is not intended for use on sequence data with N symbols; it will run but will not handle such missing data carefully.

Caution: The output is not lexicographically sorted, nor uniformly randomized.

CHAPTER

FOUR

BUILDING BEAR MODELS

4.1 bear_net

The `bear_net` submodule contains functions to train, evaluate, and deploy BEAR (and AR) models. Two example autoregressive functions are implemented in the submodule `ar_funcs`: a linear function and a simple convolutional neural network (CNN), `bear_model.ar_funcs.make_ar_func_linear()` and `bear_model.ar_funcs.make_ar_func_cnn()` respectively.

The standard workflow for training a BEAR model is to first load a file of preprocessed kmer transition counts using the `dataloader` submodule:

`bear_model.dataloader.`**`dataloader`**(*file*, *alphabet*, *batch_size*, *num_ds*, *cache=True*, *header=False*, *n_par=1*, *dtype=tf.float64*)

Load counts data into tensorflow data object.

Parameters

file [str] Location of counts data, which should be a tsv file with rows in the format: kmer_sequence counts_matrix, delimited by a tab.

alphabet [str] One of 'dna', 'rna', 'prot'.

batch_size [int] By minibatching early, the counts matrices for multiple kmers may be decoded at once.

num_ds [int] Number of columns in the count data. Ex: 3 for train, test and reference.

cache [bool, default = True] Whether or not to cache the loaded data. Increases loading speed at the cost of memory.

header [bool, default = False] Whether or not there is a header in the counts data.

n_par [int, default = 1] Number of parallel calls to turn counts matrix strings to tensors.

dtype [dtype, default = tf.float64]

Returns

data [tensorflow data object] One element of the data is a list of batch_size kmers and a counts tensor of shape [batch_size, num_ds, alphabet_size+1].

The loaded dataset can then be used to train an AR or BEAR model with the `bear_net` submodule:

`bear_model.bear_net.`**`train`**(*data*, *num_kmers*, *epochs*, *ds_loc*, *alphabet*, *lag*, *make_ar_func*, *af_kwargs*, *learning_rate*, *optimizer_name*, *train_ar*, *acc_steps=1*, *params_restart=None*, *writer=None*, *dtype=tf.float64*)

Train a BEAR or AR model using all available GPUs in parallel.

Parameters

data [tensorflow data object] Load sequence data using tools in dataloader.py. Minibatch before passing.

num_kmers [int] Total number of kmers seen in data. Used to normalize estimate of loss.

epochs [int]

ds_loc [int] Column in count data that corresponds with the training data.

alphabet [str] One of 'dna', 'rna', 'prot'.

lag [int]

make_ar_func [function] Takes lag, alphabet_size, af_kwargs, dtype and returns an ar_func. See ar_funcs submodule.

af_kwargs [dict] Keyword arguments for particular ar_func. For example, filter_width.

learning_rate [float]

optimizer_name [str] For example 'Adam'.

train_ar [bool] Whether to train an AR (True) or BEAR (False) model.

writer [tensorboard writer object, default = None]

acc_steps [int, default = 1] Number of steps to accumulate gradients over.

params_restart [list of tensorflow variables, default = None] Pass the parameter list from a previous run to restart training.

dtype [dtype, default = tf.float64]

Returns

params: list List of parameters as tensorflow variables.

h_signed [dtype] $\log(h)$ where h is the BEAR concentration parameter.

ar_func [function] The autoregressive function.

One then may evaluate the performance of the trained model:

`bear_model.bear_net.`**`evaluation`**(*data*, *ds_loc_train*, *ds_loc_test*, *alphabet*, *h*, *ar_func*, *van_reg*, *dtype=tf.float64*)

Evaluate a trained BEAR, AR or vanilla BEAR model.

Parameters

data [tensorflow data object] Load sequence data using tools in dataloader.py. Minibatch before passing.

ds_loc_train [int] Column in count data that corresponds with the training data.

ds_loc_test [int] Column in count data that corresponds with the testing data.

alphabet [str] One of 'dna', 'rna', 'prot'.

h [dtype] The h parameter from the BEAR model.

ar_func [function] A function that takes a tensor of shape [A1, ..., An, lag, alphabet_size+1] of dtype and returns a tensor of shape [A1, ..., An, alphabet_size+1] of dtype of transition probabilities for each kmer. The autoregressive function.

van_reg [float] Prior on vanilla BEAR model (Dirichlet concentration parameter).

dtype [dtype, default = tf.float64]

Returns

log_likelihood_ear, log_likelihood_arm, log_likelihood_van [float] Total log likelihood of the data with the model evaluated as a BEAR, AR or vanilla BEAR model.

perplexity_ear, perplexity_arm, perplexity_van [float] Perplexity of the data with the model evaluated as a BEAR, AR or vanilla BEAR model.

accuracy_ear, accuracy_arm, accuracy_van [float] Accuracy of the data with the model evaluated as a BEAR, AR or vanilla BEAR model. Ties in maximum model probability are resolved randomly.

To recover the concentration parameter/misspecification diagnostic h and the learned autoregressive function from the outputted list of parameters, use the `change_scope_params` function:

`bear_model.bear_net.`**`change_scope_params`**(*lag*, *alphabet_size*, *make_ar_func*, *af_kwargs*, *params*, *dtype=tf.float64*)

Redefine and get parameters of BEAR or AR distribution in given scope.

Used to to get unmirrored variables after training on multiple GPUs in parallel or to unpack a list of params into h and the autoregressive function.

Parameters

lag [int]

alphabet_size [int]

make_ar_func [function] Takes lag, alphabet_size, af_kwargs, dtype and returns an ar_func.

af_kwargs [dict] Keyword arguments for particular ar_func. For example, filter_width.

params: list List of parameters as tensorflow variables.

dtype [dtype, default = tf.float64] dtype for the ar_func and h.

Returns

params [list] List of parameters as tensorflow variables.

h_signed [dtype] $\log(h)$ where h is the BEAR concentration parameter.

ar_func [function] A function that takes a tensor of shape [A1, ..., An, lag, alphabet_size+1] of dtype and returns a tensor of shape [A1, ..., An, alphabet_size+1] of dtype of transition probabilities for each kmer. The autoregressive function.

To embed your own autoregressive function, create a new `make_ar_func_...` function using the examples `bear_model.ar_funcs.make_ar_func_linear()` and `bear_model.ar_funcs.make_ar_func_cnn()` as templates.

4.2 bear_ref

Besides standard autoregressive models like linear models and neural networks, one can also build an autoregressive model based on a reference genome. In particular, we can predict the next base by looking up the previous L bases k in the reference, normalizing the observed transition counts $\#_{\text{ref}}(k, b)$ to form a probability, and accounting for noise using a Jukes-Cantor mutation model (with error rate τ). This gives the autoregressive function

$$\tilde{f}_{k,b} = e^{-\tau} \frac{\#_{\text{ref}}(k, b)}{\sum_{b' \neq \$} \#_{\text{ref}}(k, b')} + \left(1 - e^{-\tau}\right) \frac{1}{|\mathcal{B}|}$$

for $b \neq \$$ and $\tilde{f}_{k,\$} = 0$, where $\$$ is the stop symbol and $|\mathcal{B}|$ is the alphabet size (excluding the stop symbol). When $\sum_{b' \neq \$} \#_{\text{ref}}(k, b') = 0$, we default to $\tilde{f}_{k,b} = 1/|\mathcal{B}|$. Reference genomes do not include stop symbols, and so do not

provide predictions of read length; to account for this problem in a generalized way we introduce another AR function g and combine it with the reference-based prediction,

$$f_{k,b} = \nu g_{k,b} + (1-\nu)\tilde{f}_{k,b}$$

for some $\nu \in (0,1)$. The function g is typically chosen to predict a stop symbol with probability 1; this particular function is implemented in `bear_model.ar_funcs.make_ar_func_stop()`. To train the model efficiently, we preprocess the reference sequence along with the training and testing data, such that $\#_{\text{ref}}$ forms a column of the summarized data. The submodule `bear_ref` trains this reference-based BEAR model, optimizing the hyperparameters τ and ν as well as any additional parameters in g.

`bear_model.bear_ref.`**`train`**(*data*, *num_kmers*, *epochs*, *ds_loc*, *ds_loc_ref*, *alphabet*, *lag*, *make_ar_func*, *af_kwargs*, *learning_rate*, *optimizer_name*, *train_ar*, *acc_steps=1*, *params_restart=None*, *writer=None*, *dtype=tf.float64*)

Train a BEAR or AR model based on reference transition counts using all available GPUs in parallel.

Parameters

data [tensorflow data object] Load sequence data using tools in dataloader.py. Minibatch before passing.

num_kmers [int] Total number of kmers seen in the data. Used to normalize estimate of loss.

epochs [int]

ds_loc [int] Column in count data that corresponds with the training data.

ds_loc_ref [int] Column in count data that corresponds with the reference data.

alphabet [str] One of 'dna', 'rna', 'prot'.

lag [int]

make_ar_func [function] Takes lag, alphabet_size, af_kwargs, dtype and returns an ar_func. See ar_funcs submodule.

af_kwargs [dict] Keyword arguments for particular ar_func. For example, filter_width.

learning_rate [float]

optimizer_name [str] For example 'Adam'.

train_ar [bool] Whether to train an AR (True) or BEAR (False) model.

writer [tensorboard writer object, default = None]

acc_steps [int, default = 1] Number of steps to accumulate gradients over.

params_restart [list of tensorflow variables, default = None] Pass the parameter list from a previous run to restart training.

dtype [dtype, default=tf.float64]

Returns

params: list List of parameters as tensorflow variables.

h_signed [dtype] $\log(h)$ where h is the BEAR concentration parameter.

ar_func [function] The autoregressive function.

The functions `bear_model.bear_ref.evaluate()` and *bear_model.bear_net.change_scope_params()* are analogous to the functions in `bear_net` with the same names.

CHAPTER

FIVE

EXAMPLE BEAR MODELS

We have also written scripts that will perform the above workflow - loading data, training, evaluating, and saving the list of parameters using `dill` - with parameters specified in a config file. Config files contain information about the training and testing data; parameters for training and testing; and parameters of the AR model. Example config files may be found in `bear_model/models/config_files`. Descriptions of the outputs of these scripts in given in the tutorial below.

5.1 bear_model/models/train_bear_net.py

Train and evaluate AR or BEAR models using maximal liklihood or empirical Bayes respectively. Usage:

```
python train_bear_net.py config.cfg
```

Example config files, with descriptions of the input parameters, can be found in the subfolder `bear_model/models/config_files` (*bear_lin_bear.cfg*, *bear_lin_ar.cfg*, *bear_cnn_bear.cfg*, and *bear_cnn_ar.cfg*).

5.2 bear_model/models/train_bear_ref.py

Train and evaluate AR or BEAR models that require a reference using maximal liklihood or empirical Bayes respectively. Train reference-based Bayesian embedded autoregressive models using empirical Bayes, and evaluate based on heldout likelihood, perplexity, and accuracy. Usage:

```
python train_bear_ref.py config.cfg
```

Example config files, with descriptions of the input parameters, can be found in the subfolder `bear_model/models/config_files` (*bear_stop_bear.cfg* and *bear_stop_ar.cfg*).

If the data includes a reference, the same config file may be used by either of the above scripts, with the specified AR model interpreted as g when used with `bear_model/models/train_bear_ref.py`. These scripts are easiest to use through the command line, as shown in the tutorial below.

CHAPTER

SIX

TUTORIAL

In this tutorial, we will apply BEAR models to whole genome sequencing data from the Salmonella bacteriophage YSD1. The data and the reference assembly are from Dunstan et al. (2019); the NCBI SRA record is here.

Part 1: preprocessing

First, navigate to the folder `bear_model` and download the example dataset and extract its contents (we assume throughout that you are in the `bear_model` folder).

```
wget https://marks.hms.harvard.edu/bear/ysd1_example.tar.gz
```

```
tar -xvf ysd1_example.tar.gz -C data
```

There should now be five data files in the `data` subfolder, corresponding to training sequence data (*1_train.fasta* and *2_train.fasta*), testing data (*1_test.fasta* and *2_test.fasta*), and the genome reference assembly (*virus_reference.fna*). The file *datalist.csv* lists the dataset paths, data types (all fasta), and the data groups (training, testing, and reference). We provide *datalist.csv* as input to the summary statistic script,

```
python summarize.py data/datalist.csv data/ysd1 -l 5
```

This extracts kmer transition counts up to lag 5 (inclusive). The counts for each lag will be found in the files *data/ysd1_lag_1_file_0.tsv*, *data/ysd1_lag_2_file_0.tsv*, *data/ysd1_lag_3_file_0.tsv*, *data/ysd1_lag_4_file_0.tsv*, and *data/ysd1_lag_5_file_0.tsv*. Each line consists of an individual kmer and the transition counts in each data group. Finally, before training a non-vanilla BEAR model, datasets should be shuffled. In Linux,

```
shuf data/ysd1_lag_5_file_0.tsv -o data/ysd1_lag_5_file_0_shuf.tsv
```

On a Mac, replace `shuf` with `gshuf` (you may first need to install GNU coreutils, via e.g. `brew install coreutils`). Note that shuffling is done in memory, so when using large lags on large datasets, you can use the **-mf** flag in *summarize.py* to ensure its output files are sufficiently small. A preshuffled dataset is provided in the file `models/data/ysd1_lag_5_file_0_preshuf.tsv` to ensure that part 2 of this tutorial is reproducible and can be run independently of KMC.

Part 2: training

Now we can train AR or BEAR model via maximal likelihood or empirical Bayes respectively. We will be using Config files for training three different AR models and their corresponding BEAR model can be found in the folder `bear_model/models/config_files`. You can run these examples on your own shuffled dataset from part 1 by editing the config files to set `start_token = ysd1_lag_5_file_0_shuf.tsv`; by default the config files use the example preshuffled file. Note that in these examples we fix the model lag at the small value of 5 so that it can be trained quickly; normally we would choose the lag based on maximum marginal likelihood. All 6 config files use the same training protocol and differ only in the AR functions they use (see config file section `[model]`) and whether they are BEAR or AR models (see config file entry `train_ar`). To run the examples, move to the `bear_models` directory and run the following:

Linear AR `python models/train_bear_net.py models/config_files/bear_lin_ar.cfg`

CNN AR `python models/train_bear_net.py models/config_files/bear_cnn_ar.cfg`

Reference AR `python models/train_bear_ref.py models/config_files/bear_stop_ar.cfg`

Linear BEAR `python models/train_bear_net.py models/config_files/bear_lin_bear.cfg`

CNN BEAR `python models/train_bear_net.py models/config_files/bear_cnn_bear.cfg`

Reference BEAR `python models/train_bear_ref.py models/config_files/bear_stop_bear.cfg`

Each example should not take more than a few minutes to run. The scripts each output a folder named with the time at which they were run in `out_data/logs` (two folders may already be present if you tested your installation). This output folders contains:

- A progress file that can be used to visualize the training curve using TensorBoard.
- A config file that consists of the input config file appended with the training results, in the section `[results]`. (Note that even when the BEAR model is the model that is trained, the perplexity, log likelihood and accuracy of the embedded AR model and vanilla BEAR model (BMM) are also reported; when the AR model is trained, the performance of the corresponding BEAR model with $h = 1$ is reported.)
- A pickle file with the learned hyperparameters. These hyperparameters can be recovered using the `dill` package.

The performance results from these example models should match closely the following table:

Experiment	Perplexity	Accuracy	h
Linear AR	3.99	32.9%	N/A
CNN AR	3.85	35.8%	N/A
Reference AR	3.84	36.5%	N/A
BMM	3.79	36.8%	N/A
Linear BEAR	3.79	36.8%	0.0433
CNN BEAR	3.79	36.8%	0.0119
Reference BEAR	3.79	36.8%	0.0142

A few things to note from looking at the results:

- In the paper we used a lag of 13 for this dataset, chosen by maximum marginal likelihood. Here, using a lag of 5, the perplexities are much larger and the h values are much smaller. This is due to the fact that the closest (in KL) AR model of lag 5, unsurprisingly, has a much larger perplexity than the lag 13 model, and is also much closer to the linear and CNN models.
- As demonstrated in the paper, with enough data, the relative benefit of the BEAR over the vanilla BEAR is minimal. Since the lag is only 5 in this example, the data is sufficiently large (relative to the model flexibility) to make this true. However, the BEAR still outperforms the vanilla BEAR in the fifth or sixth decimal space (not shown in table).
- The value of h is in proportion to expected misspecification of the parametric AR model. Interestingly in this simple case, unlike most example datasets in the publication, the CNN is better specified than the reference ($h = 0.0116$ versus $h = 0.0142$). Running a reference model with g set to a CNN, via `python train_bear_ref.py config_files/bear_cnn_bear.cfg`, yields an even lower $h = 0.00422$, suggesting the two models learn complementary information.
- The learned stop rate ν in the reference AR model is near 151 (not shown in table). 150 is the read length, so 1/151 transitions are stops on average.
- Embedded AR models trained in the context of a BEAR model produce better BEAR models than AR models trained using maximum likelihood (not shown in table). The same is true for AR models trained using maximal likelihood and evaluated as AR models rather than as BEAR models.

PYTHON MODULE INDEX

b

INDEX