Documentation of the CAMEOS Double-Encoding Algorithm

Tom Blazejewski, tb2595@cumc.columbia.edu

June 2019

Contents

Overview

The CAMEOS project seeks to replicate the interesting natural phenomenon of overlapping genes (two protein products translating from different frames of a single fragment of DNA) through synthetic design. In order to achieve this aim it combines a fast first-order optimization step targeting Hidden Markov Model likelihoods followed by approximate optimization of long-range interactions defined by a Markov Random Field across many stochastic initializations.

Details on the algorithm and some initial applications of the work are available in an accompanying paper.

Requirements

Environment

The documentation assumes a Unix environment. I have not tested using Windows, but so long as `hmmer` and relevant julia packages (described below) work, there should be no reason for the code to fail in this environment.

We require three further packages associated with HDF5, GZip, and HMMs. These should be widely available and correspond to the packages, `hdf5-tools`, `zlib1g-dev`, and `hmmer` in the Ubuntu package manager.

Julia

Most code associated with CAMEOS is written in `julia`, a relatively new numerical programming language. This code was originally written for v0.5 and subsequently updated to v1.0. The code continues to work well in v1.1. You can download appropriate julia binaries from https://julialang.org/.

Julia packages

The code requires several `julia` packages: `BioAlignments`, `BioSymbols`, `Logging`, `StatsBase`, `Distributions`, `JLD`, `ArgParse`. The julia package `NPZ` and the python package `tensorflow` are also used for optional local energy calculations.

If you are new to Julia, but have it in your `PATH`, the way to install these packages (assuming you have the required packages, `hdf5-tools` and `zlib1g-dev`) is to type `julia` into a terminal. Once you have entered the Julia shell, run:

```
julia> using Pkg
julia> Pkg.add("BioAlignments")
julia> Pkg.add("BioSymbols")
julia> Pkg.add("Logging")
julia> Pkg.add("StatsBase")
julia> Pkg.add("JLD")
julia> Pkg.add("Distributions")
julia> Pkg.add("ArgParse")
julia> Pkg.add("NPZ") #if running local-energy code.
julia> using BioAlignments, BioSymbols, Logging, NPZ
julia> using StatsBase, JLD, Distributions, ArgParse
#the final lines pre-compile the packages for future use.
```

Running an Example

Folder structure

We provide example data by which to try to run a project. We begin by taking a look at the contents of the `main/` directory in this repository. The folder contains a `.txt` and several `.jl` files, which contain program logic. They also contain the directories: `msas/`, `psls/`, `energies/`, `hmms/`, `mrfs/`, and `output/` as well as three fasta files: `optional_consensus.fa`, `proteins.fasta` and `cds.fasta`.

Our example considers the protein pair of *infA* and *aroB*. These folders and files contain information (with file names matching each protein) related to the pseudolikelihood (`psls`) and energies (`energies`) of natural sequences (derived from multiple sequence alignments in `msas/`), as well as Hidden Markov Model (`hmms`) and Markov Random Field (`mrfs`) parameterizations of the proteins we seek to engineer and amino acid (`proteins.fasta`) / coding sequences (`cds.fasta`) of these proteins. The `output/` folder will store program outputs.

Guidelines for creating your own versions of these files is available later in this documentation.

Running the example

As the example files are already present, running the example is simple:

```
$ julia main.jl example.txt
```

This runs the main optimization code of CAMEOS on the protein pairs defined in `example.txt`.

Looking at the structure of `example.txt`, we see that it is a single tab-delimited line with ten entries. In general we optimize two genes, referred to as A and B. The tabs, in order, are:

- Output directory: `output/`
- Gene A Name: `infA`
- Gene B Name: `aroB`
- Path to JLD MRF file for Gene A: `jlds/infA.jld`
- Path to JLD MRF file for Gene B: `jlds/aroB.jld`
- Path to HMM file for Gene A: `hmms/infA.hmm`

- Path to HMM file for Gene B: `hmms/aroB.hmm`
- Number of seeds to optimize: `100`
- Frame: always `p1`
- Number of iterations: `250`

Names for proteins A/B should concord with identifiers in `proteins.fasta` and `cds.fasta`, as well as files in `energies/` and `psls/`. They should be short as they are also used in downstream output file/folder names.

Also note that `+2` frame corresponds to switching protein A with protein B. Therefore to run in `+2`, switch the order of the proteins. The `p1` input is a placeholder for potential future development of more transparent frame options.

General outputs

CAMEOS is stochastic, so some variations in output from run to run are to be expected.

Visible outputs should be the following (in this order):

- `The random barcode on this run is: Qpib2wiY`: each run is assigned a random barcode attached to log and output files.
- `CAMEOS tensor built`: declares successful forward pass, construction of CAMEOS tensor.
- `Evaluating HMM seeds`: beginning of evaluation of solutions (we generate more than necessary then prune)
- `Beginning long-range optimization`: beginning of iterative optimization of sequences (2nd phase of algorithm)
- `Step 50 of 250...`: updates every 50 steps of progress for long-range interaction optimization.

Run-time of HMM solutions is generally fast, and depends on the lengths of the proteins being encoded. Run-time of the iterative optimization of long-range interactions tends to dominate the overall run-time, and is most influenced by the number of seeds and the number of iterations, though longer proteins are also slower to optimize due to the expense of calculating candidate pseudolikelihood values. Given the relatively small number of seeds, this example should take approximately 5-10 minutes to run on a fairly standard computer.

Several outputs are produced (BC denotes barcode):

- `all_final_fitness_BC.txt`
- `deg_BC.tbl`
- `deg_out_BC.fa`

- `deg_select_BC.fa`
- `mark_BC.tbl`
- `mark_out_BC.fa`
- `mark_select_BC.fa`
- `log_BC.txt`
- `opt_his_mat_BC.jld`
- `saved_pop_BC.jld`
- `top_twelve_BC.fa`

`log_BC.txt` records a large amount of inputs relevant to the run. It is updated during the run and can be checked over the course of optimization to see more frequent updates of iteration number and score. The `top_twelve_BC.fa` is a useful first stop for most post-optimization analyses, showing the top-3 variants in temrs of scores in either protein, as well as the top-6 variants in terms of overall score.

More information about each of these candidates can then be accessed through exploration of the `saved_pop_BC.jld`. For example, if you are interested in the 10th variant (presumably as it is indicated in the top_twelve file), you can access this variant through a julia shell.

```
julia> using JLD
julia> all_vars = load("saved_pop_BC.jld")["variants"];
julia> tenth = all_vars[10];
julia> tenth.mark_prob #pseudolikelihood for protein A
julia> tenth.deg_prob #pseudolikelihood for protein B
julia> tenth.mark_seq #protein sequence for for protein A
julia> tenth.deg_seq #protein sequence for protein B
julia> tenth.full_sequence #full CDS for both proteins A/B.
```

A brief explanation of other output files: Pseudolikelihood values for all optimized variants at end of optimization are recorded in `all_final_fitness_BC.txt`. `mark_out_BC.fa` stores all generated HMM sequences for the first "mark" protein. `mark_select_BC.fa` stores all HMM sequences selected to seed downstream long-range interaction optmization. `mark_BC.tbl` records output of HMMER scoring on these sequences. Files beginning with `deg` are defined similarly. `opt_his_mat_BC.jld` records the values of pseudolikelihoods over optimization.

Example output

In my test run of *infA* / *aroB* I see a top variant in `top_twelve_BC.fa` of:

```
>TOP_GEN_d (ind 88 ) (fit: 374.12) (deg: 327.40) (mark: 46.72)
MERWQVTTASRSLPLSLRSSLLRSSASSFRPVKRSSRTFLVVSESTASAFLPATRSSLSS
AQVAVDEVVLPSGESSKSLAVLDTVFTALLQKPHGRDTTLVALGGGVVGDLTGFAAASYQ
RGVRFIQVPTTLLSQVDSSVGGKTAVNHPLGKNMIGAFYQPASVVVDLDCLKTLPPRELA
SGLAEVIKYGIILDGAFFNWLEENLDALLRLDGPAMAYCIRRCCELKAEVVAADERETGL
RALLNLGHTFGHAIEAEMGYGNWLHGEAVAAGMVMAARTSERLGQFSSAETQRIITLLKR
AGLPVNGPREMSAQAYLPHMLRDKKVLAGEMRLILPLAIGKSEVRSGVSHELVLNAIADC
QSA
>TOP_GEN_m (ind 88 ) (fit: 374.12) (deg: 327.40) (mark: 46.72)
MAGDDSIEKSAVVVEVLPSTLFRVKLPTGQEILAHISGRLRKHRIRILAGDKVLVELSPS
GRGRGRITFRGK
```

BLASTing these proteins, we see that over the double-encoding region, the *aroB* encoding is about 50% similar to wild-type sequences while *infA* is about 76% similar to the IF-1 protein of a *Psychrobacter* homolog. It is difficult to guess *a priori* if this protein would be functional: the model may be overly-confident that it only needs to match approximately half of residues at the N-terminus of *aroB*. A full-scale attempt to double-encode the proteins would likely benefit from a library approach similar to the one used in our paper, trying several different weighings of the pseudolikelihoods of both proteins.

Looking at a top variant w.r.t. *aroB* (the "DEG") protein, we see an encoding with slightly higher local similarity (approx. 62%):

```
>TOP_DEG_d (ind 48 ) (fit: 383.45) (deg: 301.79) (mark: 81.66)
MERIVVTLGERSYPITIASGLFNEPASFLKPLKSGEQVMLVTNETLAPLYLDKVRGVLEQ
AGVNVDSVILPDGEQYKSLAVLDTVFTALLQKPHGRDTTLVALGGGVVGDLTGFAAASYQ
RGVRFIQVPTTLLSQVDSSVGGKTAVNHPLGKNMIGAFYQPASVVVDLDCLKTLPPRELA
SGLAEVIKYGIILDGAFFNWLEENLDALLRLDGPAMAYCIRRCCELKAEVVAADERETGL
RALLNLGHTFGHAIEAEMGYGNWLHGEAVAAGMVMAARTSERLGQCPAENTSRLRALLRR
CSLPVSLPSSWSPEALLPHMLRDKKVSSGSVSLVVMTSLGRSALMTSLEDALLLDTLKDC
QSA
>TOP_DEG_m (ind 48 ) (fit: 383.45) (deg: 301.79) (mark: 81.66)
MSSGEHIEVKGVVKEVLPSGQFTVELESGSTVTAHVAGQKSKFRIRVLSGDDVTGALSPY
DLTRGRITFRHS
```

This encoding targets the C-terminus of *aroB*. While the infA still shows high similarity (75%) to various bacterial IF-1 proteins, it is worth noting that the pseudolikelihood in the first example of 46.7 is very close to the mean of natural sequence infAs, whereas the score of 81.6 is more than 4 standard deviations from the mean. In general, BLAST hits can give a first glimpse of basic features of encodings but pseudolikelihoods remain a better measure of protein family fidelity.

Finally it is worth noting that running the optimization with more seeds and more iterations would likely further improve results.

Designing your own proteins

Multiple Sequence Alignments

CAMEOS fundamentally requires large and high-quality protein sequence alignments in order to succeed. Markov Random Fields generally require thousands or even tens of thousands of non-redundant sequences to be available for parameter learning. It is possible that metagenomic assemblies could serve as a useful source of extensive sequence diversity if sequences available from sources such as UniRef90 or InterPro are not sufficient.

A factor in training Markov Random Fields is that they tend to be less flexible than HMMs when considering insertions/deletions. Multiple sequence alignments should therefore be judiciously pruned ahead of MRF training, so that each column in the MSA corresponds to an expected residue in a 3-D structure. This procedure often requires careful manual curation in order to ensure that the alignment does not remain too gappy and so that important residues are not over-zealously pruned. One strategy we have found to be helpful is to consider pruning residues according to a sequence of interest (often a protein in *E. coli*). This helps to avoid situations by which the deletion of columns based on averaged statistics lead to outcomes where no protein is without insertion/deletion.

Training HMM

HMMs can be easily trained using `hmmer`. In this document we refer to files as `protein.suffix` for generality. Given a MSA `protein.msa` (in real usage, e.g. `infA.msa`) and the desire to create a HMM with path `protein.hmm` (e.g. `infA.hmm`), the commands are simply:

```
$ hmmbuild protein.hmm protein.msa
$ hmmpress protein.hmm
```

Training MRF

MRFs can be similarly easily trained using `CCMPred`. First we need to convert the multi-fasta alignment to the format expected by `CCMPred`. This can be done by removing all "header" items in a fasta file, and is most simply accomplished by running an inverse-match grep command followed by the `CCMPred` training step from which we hope to extract the `.raw` file:

```
$ grep -v ">" protein.msa > protein.ccm
$ $CCMPRED_HOME/bin/ccmpred -r protein.raw -n 100 protein.ccm protein.mat
```

We then convert this 'raw' matrix to our expected internal MRF file format: a JLD file, similar to a npy file in python.

```
$ julia $CAMEOS_HOME/prepare/convert_ccm_to_jld.jl protein.raw protein.jld
```

Summarizing pseudolikelihoods/energies

The pseudolikelihoods and energies of proteins in the targeted multiple sequence alignment are useful for optimization, so we require that these values be calculated. This can be accomplished by running the script:

```
$ julia $CAMEOS_HOME/prepare/energies_and_psls.jl protein protein.jld protein.msa
```

which will output files `psls_protein.txt` and `energy_protein.txt` in the appropriate directories. Note the first argument, the name of the protein. This avoids needing to infer this from jld/msa paths which may be absolute and hard to parse.

Optional: These calculations are computationally intensive. If you wish to speed up the execution of these calculations and have access to a GPU, we provide tensorflow code in the file `tensorflow_energies_psls.py` to speed up computation significantly. Use of the code requires running the `.jld` file through the `convert_jld_to_npy.jl` program (see below).

Protein Sequences and CDS

Finally, because CAMEOS products are generally composed of overlapping- and non-overlapping regions, we require full protein and CDS sequences for each targeted protein. These should be incorporated into their respective fasta files in the main directory, with headers `>protein` where protein is the name of the targeted protein.

CAMEOS has some robustness against mismatches between MRF/HMM models and wild-type proteins, but it is best for the proteins in these files to match the pruning pattern of the multiple-sequence alignment.

Optional: Local energies

Long-range interactions are not uniformly distributed in most proteins. It may therefore be useful to consider the 'density' of long-range interactions in order to select specific regions that are to be double-encoded (otherwise regions are selected based only on HMM likelihoods which will neglect epistatic considerations). This is achieved by only considering interactions within a specified window of proteins, and by assessing the strength of these interactions across many sequences in an alignment.

This operation, if replicated across multiple window sizes, can be fairly computationally intensive. We therefore provide a tensorflow script able to utilize GPU capabilities to speed up the operation. We also only compute the information for a subset of 2000 sequences. This parameter can be modified in the original script.

The script requires conversion of the jld file to npy files (in turn requiring a new julia package `NPZ`):

```
$ julia convert_jld_to_npy.jl protein.jld
```

The tensorflow script should be edited in a text editor as several parameters have to be adjusted (see code for details, near bottom of file):

```
$ python local_energies_tf.py
```

This produces .npy files containing a matrix where each row is a position and columns are measurements of local energy across tested variants. In our paper we visualize the mean/standard deviation of these measurements across positions in a protein.

Running your protein pairs

With this preparation complete, you can now run the CAMEOS code on your gene pairs of interest by creating a new command file, `newrun.txt` containing a line like:

```
out_folder/ A B A.jld B.jld A.hmm B.hmm 500 p1 1000
```

If you would like to change the region being double-encoded, simply add the region of interest to the line. Regions are defined in terms of the final position in the protein, so if protein A is 150 aa and protein B is 50 aa and you want to encode protein B starting at positions 25-30, this can be indicated by:

```
out_folder/ A B A.jld B.jld A.hmm B.hmm 500 p1 1000 150 150 75 80
```

Where 150/150 is the length of protein A and 75/80 are 25 + length of protein B, 30 + length of protein B.

Running many pairs

Many pairs and positions of pairs can be run serially simply by adding lines to the command file: CAMEOS will iterate through each line in the presented file.

With a very simple approach, we can run multiple jobs in parallel by running multiple julia instances at once:

```
julia main.jl newrun.txt --num 0 --threads 4 &
julia main.jl newrun.txt --num 1 --threads 4 &
julia main.jl newrun.txt --num 2 --threads 4 &
julia main.jl newrun.txt --num 3 --threads 4 & wait
```

The code will then read every 4th line with offset `num`, allowing 4 jobs to be run in parallel across different threads.

A note if trying much larger-scale experiments: I've observed `hmmer` experience difficulties with very large numbers of threads. Increasing the number of system processes available by running, `ulimit -u 4096` seems to help. For running multiple threads the code also randomly staggers runs to try to avoid mutliple threads trying to create the same directories at the same time. This could be further tuned for robustness.

Conclusion

Thank you for your interest in this project, we hope this documentation serves as a useful introduction to our work. This project is the product of a large amount of work from Hsing-I Ho, Harris Wang, and Tom Blazejewski. We hope that the code will allow you to achieve your double-encoding aims, and let us know on Github or by e-mail if you encounter bugs or issues not covered in the documentation.