README.md

ABC-DLS (Approximate Bayesian Computation with Deep Learning and Sequential monte carlo)

This software is a python pipeline where you can use simulated summary statistics to predict which underlying model can better explain the observed or real-world results (classification) and can guess which parameters can produce such results (parameter estimation) with the help of Approximate Bayesian Computation (ABC), Deep Learning (using Tensorflow Keras Backend) and Sequential Monte Carlo (SMC). The whole method is written in python and easy to read and can be accessed entirely through the command line, therefore knowing python is not needed. Although it will be helpful to know python and R as some of the packages here used are based on those languages.

Getting Started

To download the package, either click the download button in https://github.com/mayukhmondal/ABC-DLS or use:

```
git clone https://github.com/mayukhmondal/ABC-DLS
cd ABC-DLS
```

The codes are written in python3 (>=python3.6.9). This programme comes with several dependencies:

- numpy
- scikit-learn
- joblib
- pandas
- h5py
- rpy2
- r-essentials
- tzlocal
- tensorflow
- keras

The easiest way to install all of these dependencies is using conda. To install conda please visit anaconda. After installing conda (remember to install python 3), use:

```
conda install --file requirements.txt
```

or for last tested version

```
conda env update -f requirements.yml
```

or if you want to make a different environment:

```
conda create --name ABC-DLS --file requirements.txt  python=3
conda activate ABC-DLS
```

same but for last tested version. Use it in case you getting conflict between package versions.

```
conda env create -f requirements.yml --name ABC-DLS
conda activate ABC-DLS
```

Please wait a little bit as it can take a long time to install all the dependencies. After installing all the dependencies, you can just run either

```
python src/Run_Classification.py --help
```

or

```
python src/Run_ParamsEstimation.py --help
```

or

```
python src/Run_SMC.py --help
```

For Model selection, parameters estimation and parameter estimation using SMC, respectively. The first time you run, it will also try to install abc from r package manager (automatically). Please see examples/Examples.md for a detailed guide how to use the codes. Right now, this code is written and checked in the linux system. I can not guarantee it will work on other systems, but you are welcome to try.

Installation Issues

In case you try to install it to an already existed conda environment that already has R (r-base), it can conflict with the rpy2 when it tries to automatically download an abc package from R saying abc package does not exist. In that case, create a new environment.

Citation

Revisiting the out of Africa event with a novel deep learning approach

Francesco Montinaro, Vasili Pankratov, Burak Yelmen, Luca Pagani, Mayukh Mondal
bioRxiv 2020.12.10.419069; doi: https://doi.org/10.1101/2020.12.10.419069

Contact

The code is maintained by Dr. Mayukh Mondal. In case you need further assistance please contact mondal.mayukh@gmail.com

examples/Examples.md

ABC-DLS Examples

This software is a python pipeline where you can use simulated summary statistics to predict which underlying model can better explain the observed or real-world results (classification) and can guess which parameters can produce such results (parameter estimation) with the help of Approximate Bayesian Computation (ABC), Deep Learning (using Tensorflow Keras Backend) and Sequential Monte Carlo (SMC). The simulations are preferably coming from pop genome simulators i.e. ms, msprime, FastSimcoal etc. but not necessarily bounded only by that. The whole method is written in python and thus easy to read. It can be accessed entirely through the command line, therefore knowing python is not essential, though it will be helpful to know python and a little bit of R as some of the packages in the pipeline is based on those languages. Remember, both classification and parameter estimation will create files in the current working directory or a directory mentioned in the command. Thus you cannot run multiple runs together in the same folder as it would conflict with other runs. Either run one code at a time or run them in different folders using the --folder option.

Classification and Model Selection

This part explains how to choose the best model to explain the observed data (or, in this case, real sequenced data). This programme only accepts csv files (can be zipped) as an input. The simulations (i.e., ms, msprime, FastSimcoal etc.) should be done elsewhere to produce summary statistics (ss) csv files (please see SFS for more details). One example of ss can be Site Frequency Spectrum (SFS). However, any ss, which can be represented in a single row with a similar number of columns independent of parameters or demography, can be used. The csv files should be written like this:

- Every row denotes one simulation under the model.
- First few columns should be the parameters that created the summary statistics.
- Everything else should be as the ss, which would be used by TensorFlow (TF) to differentiate between models or predict back the parameters.
- The files should have a header.

You can look inside the examples/*.csv.gz files to get an idea.

```
N_A,N_AF,N_EU,N_AS,...,0_0_0,0_0_1,0_0_2,0_0_3,0_0_4,...
14542.382466372237,119646.25929867383,75043.3425995741,103496.95227233913,...,0.0,676743.0,128199.0,47163.0,19450.0,...
14780.566576552284,20743.142386406427,90821.23078107052,117292.89382816535,...,0.0,609543.0,132621.0,54121.0,23711.0,...
15068.855032319485,71129.50749663456,71222.94672045513,119242.37644455553,...,0.0,611102.0,132722.0,57052.0,27447.0,...
14533.876139703001,25492.550958201846,78599.68927419234,74284.75369536638,...,0.0,781011.0,169080.0,78278.0,43865.0,...
14620.827267084273,92068.73287607494,99109.69362439432,140694.91698723484,...,0.0,632596.0,116719.0,44490.0,19010.0,...
```

In principle, you do not need parameter columns for the classification part, but we kept the file format similar to the later part where parameters are required.

```
python  src/Run_Classification.py --help
```

There are 5 different part of the methods:

Methods	Helps
Pre_train	To prepare the data for training in Neural Network (NN)
Train	The training part of the NN. Should be done after Pre_train part
CV	After the training only to get the result of the cross-validation test. Good for unavailable real data
After_train	This is to run the ABC analysis after the training part is done
All	The whole run of the ABC-DLS for classification from first to last. It will run all of the above steps together.

Pre_train

This is the pre training part. Where the data is prepared for the NN.

```
python src/Run_Classification.py Pre_train examples/Model.info --scale
```

- input: examples/Model.info
  It can be any text file that has all the demographic models that we want to compare together. Every line is denoted for one demographic simulation csv file. We should also denote the number of columns that are present in that file for the parameters. It will remove those columns from the files as they are not ss (parameters in this case) for the comparison.
  It should look like:

```
<Model1.csv.gz> <param_n>
<Model2.csv.gz> <param_n>
```

- --scale
  This option would scale the SFS data per column. So that all the data inside a column should be within 0-1. This strategy improves the prediction substantially.

This command will create in total three files. x.h5 (this is for ss), y.h5 (models names in integer format) and y_cat_dict.txt (models name and their corresponding integer number). These first two files will be needed to run the NN training for TF. The last part will be used later.

Train

The next part is to run the training part by NN. This command will train the NN to differentiate between models.

```
python src/Run_Classification.py Train --demography src/extras/ModelClass.py --test_size 1000
```

This command will train the model.

- --nn src/extras/ModelClass.py
  ABC-DLS has a default training neural model. But it is impossible to predict which model should be better for the input data. Thus, we can define custom made model cater to our own need. One such example is src/extras/ModelClass.py. Here, we put very few epochs (only 20) to get faster results. More is merrier, of course. The *.py should have a definition name ANNModelCheck, which should return the trained model (after model.fit) and has two inputs, x and y. Example:

```
from tensorflow.python import keras
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import *

def ANNModelCheck(x, y):
    """
    your own set of code for tensorflow model. can be both Sequential or Model class (functional API). check
    src/extras/*.py to have an idea. also check https://keras.io/ to understand how to make Keras models
    """
    model = Sequential()
    model.add(...)
    ...
    model.add(Dense(y.shape[1], activation='softmax')) # this line is important for classification
    # Model Class can also be used. Example:
    # x_0 = Input(shape=(x.shape[1],))
    # x_1 = Dense(128, activation='relu')(x_0)
    # x_1= Dense(y.shape[1], activation='softmax') (x_1)
    # model = Model(inputs=x_0, outputs=x_1)
    model.compile(...)
    # we found for classification in model.complile loss=keras.losses.categorical_crossentropy, optimizer='adam'; these
    # two gives the best results
    model.fit(x, y,...)
    return model
```

- --test_size 1000
  This option will keep the last 1000 lines left for the test data set and be used for ABC analysis. It has already shuffled data in the previous step. Thus, you would expect nearly an equal number of simulations for every demography in the last 1000 lines.

This command will save the neural model as ModelClassification.h5, which we can use later.

CV

The next will be calculating the CV error to see if our NN can differentiate between models. This step is essential if we do not have the observed data. If you have the observed data in hand, use the next part (After_train).

```
python src/Run_Classification.py CV --test_size 1000 --tolerance 0.01
```

- --test_size 1000
  This option will define the number of simulation rows (in total) used for the test data set. This test data set has never been seen by the neural network, which is useful for seeing if your model is overfitting. Only this test data set will be used for ABC (r package) analysis.
- --tolerance .01
  The ABC analysis needs this parameter to tell how much tolerance your model can have.

This command will print out the confusion matrix as well as save the CV.pdf where we can understand the power of NN to differentiate between models.

After_train

This part is similar to the CV part, but it has the observed file together in the step. Thus can be used to see which demographic model can better explain the observed data.

```
python src/Run_Classification.py After_train --test_size 1000 --tolerance 0.01 --ssfile examples/YRI_FRN_HAN.observed.csv --frac  4.6367
```

- --test_size 1000 and --tolerance .01
  Same as above for CV
- ssfile examples/YRI_FRN_HAN.observed.csv
  To define the observed csv file. Here we put YRI_FRN_HAN (Yoruba, French and Han Chinese) from the High Coverage HGDP data SFS file as SS.
- --frac 4.636757528 This option will define a fraction that has to be multiplied with observed data if the length of the simulated region does no match with observed or real data. For example, I have simulated 3gbp regions per individual in this particular case, but the real data comes after filtering around 647mbp region. To make it equal, I have to multiply the observed data with (3gbp/647mbp) or 4.636757528
- --csvout
  If you are happy with all the results, you can use csvout. This option will remove .h5 files to free up space and produce csv files, which can be used to improve the results further using R_abc if necessary. As all the commands of abc is not supported here in ABC-DLS directly.

This command will print out (including the CV part) which underlying model better explains the observed data. It will also print out the goodness of fit to see if our observed model predicted by NN comes naturally under all the distribution of such model. If you use csvout, it will additionally output model_index.csv.gz (all the model indexes), ss_predicted.csv.gz (prediction from simulated ss by NN) and ss_target.csv.gz (prediction of the observed or real data).

All

In case, rather than doing it separately, we can do all these together in one command.

```
python src/Run_Classification.py All --test_size 1000 --tolerance 0.01 --ssfile examples/YRI_FRN_HAN.observed.csv --nn src/extras/ModelC
```

- --folder
  This option will run the whole stuff inside a folder so that it does not create a lot of files in the current directory.

It will produce the same files as previously but all of them together. If we do not use --chunksize, it will create x_test.h5 and y_test.h5 (of course, if we use csvout it will be deleted) instead of x.h5 y.h5 as it will keep the training part inside the RAM itself. If you reach memory error, please use chunksize, which will be relatively slower but do not have any upper limit for the file size.

Optional

We can easily use this result in R to further our analysis:

```
library(abc)
ss_predict=read.csv('ss_predicted.csv.gz')
target=read.csv('ss_target.csv.gz')
index=as.vector(factor(as.matrix(read.table('model_index.csv.gz',header=TRUE))))
cv4postpr(as.vector(index),as.matrix(ss_predict[,-1]),nval=1000,method='rejection',tols=c(.01,.1))
```

To see with different amount of tolerance level and nval how the abc analysis changes.

Parameter Estimation

Here, we try to predict the parameters which can explain the observed results for a given model. This part will follow similarly to the previous steps. But instead of multiple files for different demography, it will only use one of the files (the very first) for parameter estimation (for a given model). It has a similar structure like Classification: All, Pre_train, Train, CV, After_train.

Pre_train

As the classification part, it will prepare the data for training.

```
python src/Run_ParamsEstimation.py Pre_train examples/Model.info --scale b
```

- input: examples/Model.info
  Exactly like the classification part. But if the Model.info file has multiple lines, it will only use the first line for the parameter estimation. --scale b If the data (both x, which is the ss and y, which is the parameters) is not already scaled, it will be scaled. It will be scaled using MinMaxscaler, which means all the numbers will be within 0 to 1 per column.

It will produce x.h5 and y.h5 like previously. The main significant difference is for y.h5. Whereas in the classification part, it only needed model names; here, it will use parameters. It will also save params_header.csv to know the name of the parameters.

Train

Same as the classification part, this will train the model. Unlike the classification, it takes many epochs and a complicated NN model to get high scores for accuracy (using the SMC part before further increases the prediction power with more precision).

```
python src/Run_ParamsEstimation.py Train --nn src/extras/ModelParams.py --test_size 1000
```

- --demography src/extras/ModelParams.py
  Although there is a default method present in ABC-DLS (meaning python src/Run_ParamsEstimation.py Train --test_size 1000 will also work), we can give a model from outside. Here we kept 100 epochs to make it faster. *.py must have a def name ANNModelParams. We can decide the number of epochs and other stuff inside that definition. The structure of the file is very similar. Only ANNModelParams instead of ANNModelCheck and the NN output is linear (which is default for keras) instead of softmax. Everything else should be done as your model prefers. Example:

```
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import *


def ANNModelParams(x, y):
    """
    your own set of code for tensorflow model. can be both Sequential or Model class (functional API). check
    src/extras/*.py to have an idea. also check https://keras.io/ to understand how to make Keras models
    """
    model = Sequential()
    model.add(...)
    ...
    model.add(Dense(y.shape[1])) # this line is important for parameter estimation
    # Model Class can also be used. Example:
    # x_0 = Input(shape=(x.shape[1],))
    # x_1 = Dense(128, activation='relu')(x_0)
    # x_1= Dense(y.shape[1]) (x_1)
    # model = Model(inputs=x_0, outputs=x_1)
    model.compile(...)
    # we found for parameter estimation in model.complile, loss='logcosh' and optimizer='Nadam', these two gives the
    # best results
    model.fit(x, y,...)
    return model
```

- --test_size 1000
  As above, it kept 1000 samples for later use. All the other samples are used for the training part.

This command will save the model as ModelParamPrediction.h5, which later can be used for prediction and stuff.

CV

To calculate the cross-validation error of the parameters.

```
python src/Run_ParamsEstimation.py CV --test_size 1000 --tolerance .01 --method loclinear
```

- --test_size 1000 --tolerance .01
  Same as clasiification.
- --method loclinear
  This option is used to tell which method to be used for CV. We found that generally, loclinear is good for CV. On top of that, if we use ABC-DLS cv method will either produce the CV independently per column for NN prediction (when using loclinear or rejection) or calcualte CV by all the columns together by NN prediction (when using neuralnet or rejection).

It will either produce nnparamcv.pdf (for using loclinear), nnparamcv_together.pdf (for using neuralnet), or both (for rejection) and print the cv error table in the terminal. It will also produce a correlation matrix for prior and posterior. This matrix is important to see if some parameters are becoming more correlated than the prior, which might be the drawback of the ss, or the NN used by TF.

After_train

After everything is done, we can use the After_train to use the ABC analysis.

```
python src/Run_ParamsEstimation.py After_train --test_size 1000 --tolerance .01 --method loclinear --csvout --ssfile examples/YRI_FRN_HAI
```

This command will calculate both the CV part as well as will compare it with the observed data. The command will produce paramposterior.pdf to see the prior vs posterior. It will also create the same csv file as before, but instead of model_index.csv.gz will generate params.csv.gz. Inside those files, there will be necessary information for the parameters.

All

To put all these parts together, we can use:

```
python src/Run_ParamsEstimation.py All --nn src/extras/ModelParams.py --test_size 1000 --tolerance .01 --method loclinear --csvout --ssf
```

It will produce a similar result but running all the commands together.

Optional

We can use further our analysis in R:

```
library(abc)
params=read.csv('params.csv.gz')
ss=read.csv('ss_predicted.csv.gz')
target=read.csv('ss_target.csv.gz')
res=abc(target = target,param=params,sumstat = ss,tol=.01,method='neuralnet',transf = 'log')
summary(res)
plot(res,param=params)
```

This command will transform the parameter values in log scale. Thus, we can calculate the distance much more precisely.

Parameter Estimation by SMC

Now Parameter Estimation by ABC-DLS is good, but what if we want to do it recursively. First and foremost, we need to understand why doing parameter estimation recursively is better in this case. For example, think it like this: before the training, we did not know the amount of admixture from the first population to the second population (suppose the actual amount is 30%). As our priors are 10%-90%, it can be 10%, 20%, .. and 90%. Anything is possible. So we run every possible admixture amount and NN learns how the ss should look under 10%,20%..90% admixture amount. We use the real/observed data, suppose it predicted the amount is 20-50% (posterior). Now we can only concentrate on simulations from 20-50% admixture as there is no need to make the NN learn how the ss behaves in those extreme conditions (<20% and >50%) when they are unlikely. NN now can specialize in much smaller deviated ss, making it much powerful for prediction. Of course, we can simulate an infinite number of lines to make the NN learn from that. Instead, we are making the NN learn recursively for the amount we think is accurate and by doing that, we are making it much more specialized. The simplified idea is to get the minimum and maximum value for every parameter as a posterior and use that posterior as a prior to create a new set of simulations and repeat it (aka Sequential Monte Carlo or SMC, which is also sometimes called Particle Filter). This recursion should be done till convergence is reached. In this case, when dec (decrease) of every parameter is more than 95% (default value), we can assume we have acquired enough convergence and the NN now cannot make any more improvement.

$$dec = \frac{Posterior_{max} - Posterior_{min}}{Prior_{max} - Prior_{min}}$$

To run the SMC for Parameter Estimation for a single time:

```
python src/Run_SMC.py All --folder SMC --nn src/extras/ModelParamsTogether.py --test_size 1000 --tolerance .05 --csvout --ssfile example
```

The code is similar to the parameter estimation part. Some added changes make it more efficient for recursion (removing most of the extra tests and graphs and only producing the range).

- --demography src/extras/ModelParamsTogether.py
  The format is slightly different than the Parameter Estimation. The idea is train and test are send together for the training part to make it more efficient. As it is a recursive method, wasting of simulations does not make sense. In case you want, you can use the default NN, which works most of the time. In case you want to use your own NN model, you have to follow this format:

```
from tensorflow.python.keras.models import Sequential
from tensorflow.python.keras.layers import *
import numpy
# from tensorflow.keras.callbacks import Callback, EarlyStopping, ModelCheckpoint, ReduceLROnPlateau

def ANNModelParams(x, y):
    # main difference from Parameter Estimation. Sending both (_train,_test) together as tupple
    x_train, x_test = x
    y_train, y_test = y
    #This part is same as before
    model = Sequential()
    model.add(...(input_shape=(x_train.shape[1],)))
    ...
    model.add(Dense(y_train.shape[1]))
    #example
    # model.add(GaussianNoise(0.05, input_shape=(x_train.shape[1],)))
    # model.add(Dense(256, activation='relu'))
    # model.add(Dense(128, activation='relu'))
    # model.add(Dense(64, activation='relu'))
    # model.add(Dense(32, activation='relu'))
    # model.add(Dense(y_train.shape[1]))
    model.compile(...)
    #example
    # model.compile(loss='logcosh', optimizer='Nadam', metrics=['accuracy'])
    model.fit(x_train, y_train, ...,validation_data=(numpy.array(x_test), numpy.array(y_test)))
    # example
    # adding an early stop so that it does not overfit
    # ES = EarlyStopping(monitor='val_loss', patience=100)
    # checkpoint
    # CP = ModelCheckpoint('Checkpoint.h5', verbose=0, save_best_only=True)
    # Reduce learning rate
    # RL = ReduceLROnPlateau(factor=0.2)
    # model.fit(x_train, y_train, epochs=5, verbose=0, shuffle="batch", callbacks=[ES, CP, RL],
    #           validation_data=(numpy.array(x_test), numpy.array(y_test)))

    return model
```

- --csvout
  This option will keep the simulations that are within the new (posterior) range of parameters. Thus can be reused for another round(s) of iteration.
- --decrease 0.95
  This option is the amount of decrease necessary to regard it as a genuine improvement. Because we are choosing the top 5% for the parameters in ABC, we always expect that the posterior range would be smaller than prior (even though the NN do not have power to predict that). To remove such estimation, we use this 95% filter, which means if the posterior is not less than 95% of the prior, we will regard the prior range as the posterior range. This filtering is essential as we are recursing so that the ranges do not decrease incorrectly.

- --increase .01
  There is always a chance when some decrease of range happened in one cycle, it missed the actual target value (suppose your actual introgression amount is 3%, but it was predicted in a cycle to be 1-2% wrongly). We use this parameter to get back the true introgression in the subsequent cycle. This option will increase the distance between the lower and upper limit by 1% (if 0.01 was used). So in a sense, you can treat increase and decrease two opposing forces. The decrease will shorten the distance between the upper and lower range, whereas the increase will broaden it up. After multiple cycles, they (increase and decrease) together generally reach a convergence. But remember to put the increase much lower than the decrease (typically five times lesser than 1-decrease). If not, it can be stuck in an infinite loop.
  $$Posterior_{min} = Prior_{min} - (Prior_{max} - Prior_{min}) \times \frac{increase}{2}$$
  $$Posterior_{max} = Prior_{max} + (Prior_{max} - Prior_{min}) \times \frac{increase}{2}$$
- --hardrange examples/hardrange.csv
  If the increase was not used, this file is not required as in every cycle the distance between lower and upper limit can only go lower. But in case of increase is used, the range can grow bigger and sometimes come to a point where the range does not make sense anymore (for example, admixture amount more than 100%), or the range is outside of what is your prior belief. Thus it is a good idea to give the starting range as a hardrange file so that your simulations will always be within that limit of starting range. Please follow the format in the examples/hardrange.csv file.

It will printout the Posterior range (which has information from ABC minimum and maximum range) and if hardrange is given it will also print out log of mean range decrease (lmrd). lmrd is a easy way to understand how much decrease or improvement you have gotten in the posterior over the hardrange or starting range. It will also save a new file called Newrange.csv, which would have information about the posterior range.

Recursion

If we cannot do the recursion, there is not much difference between Parameter Estimation and Parameter Estimation with SMC. SMC part is a subset and efficient version of the normal parameter estimation part for a single iteration. ABC-DLS is, in principle, meant for any ss (not only SFS). All the possible ways of producing ss can't be written here. Secondly, the production of ss files takes time and it is difficult to run the code for the production of ss on a single computer. It would help if you had a cluster and a pipeline (for example, snakemake), submitting multiple ss files in parallel. Nonetheless, here we will give an idea of how to do it but its reader discretion how to implement such a pipeline.

Iterations

```
do while any parameter imp < 0.95
    Produce the prior parameters with in some range
    Produce SS from those parameters (heavily parallelize here)
    Merge parameters and their corresponding ss together so it can be used in ABC-DLS
    python src/Run_SMC.py ..
    remove unimportant files
```

You can look at src/SFS/SFS_Examples.md to have an idea how to do it.

Good Practices

- Never believe in one run of ABC-DLS. From my experience, generally the results do not change much under different conditions but as NN is a black box approach, it is always better to be sure than sorry. You can run several separate runs (from pre-train) with several different neural models (you can find some in src/extras/ folder) and see if it reaches the same outcome. After training, also use differently observed ss (after train) files give identical results. One example might be to use a different mask strategy than what is used here, like mappability mask (using snapable programme) ss files and compare the results coming from them. The same things can also be achieved by producing the same observed ss from different individuals or using bootstrap results from the same individuals (in case there are no other sequences available).
- Take care of overfitting by checking accuracy in training data set vs. test data set. If training accuracy is very high compared to the test data set, try to run a smaller number of epochs or use more data to train. If you are using less data than needed, your train data set accuracy will diverge from test data set accuracy very early on (<50 epochs). This outcome suggests more data might improve your training. On the other hand, more data is always better and we can simulate more and more data effortlessly. In principle, memory is not a problem for ABC-DLS as the code is implemented in hdf5 format. Thus you have unlimited memory (in principle). But take care, as more data also means it will take more time to converge. As a rule of thumb, we found that 2k (1k for training and 1k for ABC) simulations for the classification, 60k (50k for training and 10k for ABC) simulations for the parameter estimation and 20k (10 k for training and 10k for ABC) simulations for SMC approach are generally enough.
- Use migrations (under island model) cautiously till a better method (or ss) to calculate migrations is found. Although this approach gives the freedom to use migrations, we should use it moderately. We discovered that migrations make the result less accurate (at least in the current form SFS + Neural network) and significant amount of migrations between populations has demonstrated to change the underlying tree of demographic history and thus the interpretation of the whole model itself. Migrations can affect the result indirectly, which is not easy to understand. Try to check if your model gives the same result with or without migrations. If not, revisit your model without migration, instead of believing your model with migrations.
- Remember garbage in garbage out principle. If you use nonsense data as an input, you will get a nonsense result as an output. Although by using ABC method, it is easier to catch such a situation (as ABC gives posterior distribution rather than a single number) but it is not full-proof. On top of it, NN is a black-box approach. Thus, it is sometimes nearly impossible to catch such mistakes (NN will learn anything if you force it to learn). The suggested direction will be to start from an already known and accepted result. See if you get similar results and then try to make more complex models on top of it.

src/SFS/SFS_Examples.md

ABC-DLS with SFS Examples

In principle, ABC-DLS is meant for any summary statistics (ss) that can be used to predict the parameters. But unfortunately, it is impossible to give examples for every possible ss. On the other hand, it is harder to understand and implement if examples are not provided (especially for SMC part). Here we used one of the ss to explain how to implement such a pipeline for population genomics background. Joint Site Frequency Spectrum (SFS) was shown to be good ss for this kind of approach. Although they are good but do not mean they are sufficient, ABC-DLS should not be shackled by only using SFS but this is a good starting point.

Installation

To run and create SFS, we need some new packages. I divided the packages from ABC-DLS default packages as SFS are not necessary for ABC-DLS and more packages mean more dependency hell. The packages needed on top of the previous packages are:

- scikit-allel
- msprime
- snakemake

You can install as previously (of course in the same environment as ABC-DLS):

```
conda install -c conda-forge -c bioconda --file src/SFS/requirements.txt
```

or last tested version:

```
conda env update -f src/SFS/requirements.yml
```

VCF to SFS

First, we need a real or observed ss, which can be produced from a vcf file. Before using the vcf file, we need some filters and information so that it can be used to create SFS.

- filters: Everybody has their strategy of filtering. But the strategy I generally use is the following (for explanations for the commands used, please see the respective software sites vcftools and bcftools.):

```
vcftools --gzvcf  <in.vcf.gz> --max-missing 1.0 --remove-indels --min-alleles 2 --max-alleles 2 --mac 1 --keep <in.pop> --stdout --recode
bcftools index <out.vcf.gz>
```

- Ancestral allele: We need to add the alleles' ancestral information in the INFO tag of the vcf file. You can use bcftools annotate to add that information if you have the ancestral allele file.

After we have our filtered and annotated vcf file (you can look for an example in examples/Examples.vcf.gz, we can convert it to an SFS file using:

```
python src/SFS/Run_VCF2SFS.py --popfile Input.tsv --sfs_pop YRI,FRN,HAN examples/Examples.vcf.gz
```

This command will create an Examples.csv. This file can be used as observed SFS for further analysis.

- --popfile Input.tsv
  This file should have information on populations per individual. The first column should be the individual name present in the vcf file and the second column should be the population name in a tab-separated format.
- --sfs_pop YRI,FRN,HAN
  This option used to give in which sequence the sfs should be produced as SFS (Pop1, Pop2, Pop3) != SFS(Pop2, Pop1, Pop3). Where Pop are populations.
- examples/Examples.vcf.gz is the vcf file.

Creating Uniform Priors from range

Before running the simulations, we need to create priors for the parameters on which simulations can run. The priors can be created by different methods and from different distributions. But one of the most used scenarios is to produce the priors from a uniform distribution with a given range for minimum and maximum values. Here we took the example from a well-known model (Gutenkunst et al. 2009, Gravel et al. 2013).
To produce uniform distributions of parameters, you can use:

```
python src/SFS/Run_Range2UniParameters.py --upper 25e3,2e5,2e5,2e5,1e4,1e4,1e4,80,320,700,50,50,50,50 --lower 5e3,1e4,1e4,1e4,500,500,50(
```

This command will create csv file whose every row denote different run for simulations and the columns represent various parameters with the given range:

Parameters	Upper Limit	lower Limit
N_A	25,000	5,000
N_AF	200,000	10,000
N_EU	200,000	10,000
N_AS	200,000	10,000
N_EU0	10,000	500
N_AS0	10,000	500
N_B	10,000	500
T_EU_AS	80 ky	15 ky
T_B	320 ky	5 ky
T_AF	700 ky	5 ky
m_AF_B	50×10^{-5}	0
m_AF_EU	50×10^{-5}	0
m_AF_AS	50×10^{-5}	0

Parameters	Upper Limit	lower Limit
m_AF_EU	50×10^{-5}	0

where ky is kilo years

Prior to SFS

Simulation (msprime)

Again SFS can be created by lots of other methods but here, we have used only msprime (which is fast enough as we need a lot of simulations). For the simulations, we used a previously well-known model from msprime itself with slight changes:

```
import math

import msprime
import numpy
def OOA(params, inds, length=1e6, mutation_rate=1.45e-8, recombination_rate=1e-8, replicates=300):
    (N_A, N_AF, N_EU, N_AS, N_EU0, N_AS0, N_B, T_EU_AS, T_B, T_AF, m_AF_B, m_AF_EU, m_AF_AS, m_EU_AS) = params
    (n1, n2, n3) = inds

    T_EU_AS, T_B, T_AF = numpy.array([T_EU_AS, T_B, T_AF]) * (1e3 / 29.0)
    m_AF_B, m_AF_EU, m_AF_AS, m_EU_AS = numpy.array([m_AF_B, m_AF_EU, m_AF_AS, m_EU_AS]) * 1e-5
    r_EU = (math.log(N_EU / N_EU0) / T_EU_AS)
    r_AS = (math.log(N_AS / N_AS0) / T_EU_AS)
    population_configurations = [
        msprime.PopulationConfiguration(
            sample_size=n1, initial_size=N_AF),
        msprime.PopulationConfiguration(
            sample_size=n2, initial_size=N_EU, growth_rate=r_EU),
        msprime.PopulationConfiguration(
            sample_size=n3, initial_size=N_AS, growth_rate=r_AS)
    ]
    migration_matrix = [
        [0, m_AF_EU, m_AF_AS],
        [m_AF_EU, 0, m_EU_AS],
        [m_AF_AS, m_EU_AS, 0],
    ]
    demographic_events = [
        # CEU and CHB merge into B with rate changes at T_EU_AS
        msprime.MassMigration(
            time=T_EU_AS, source=2, destination=1, proportion=1.0),
        msprime.MigrationRateChange(time=T_EU_AS, rate=0),
        msprime.MigrationRateChange(
            time=T_EU_AS, rate=m_AF_B, matrix_index=(0, 1)),
        msprime.MigrationRateChange(
            time=T_EU_AS, rate=m_AF_B, matrix_index=(1, 0)),
        msprime.PopulationParametersChange(
            time=T_EU_AS, initial_size=N_B, growth_rate=0, population_id=1),
        msprime.PopulationParametersChange(
            time=T_EU_AS, growth_rate=0, population_id=2),
        # Population B merges into YRI at T_B
        msprime.MassMigration(
            time=T_B + T_EU_AS, source=1, destination=0, proportion=1.0),
        msprime.MigrationRateChange(time=T_B + T_EU_AS, rate=0),
        # Size changes to N_A at T_AF
        msprime.PopulationParametersChange(
            time=T_AF + T_B + T_EU_AS, initial_size=N_A, population_id=0)]
    geno = msprime.simulate(
        population_configurations=population_configurations,
        migration_matrix=migration_matrix,
        demographic_events=demographic_events, length=length, mutation_rate=mutation_rate,
        num_replicates=replicates,
        recombination_rate=recombination_rate)
    return geno
```

The notable differences here are with the scaling of events and migration rates. msprime uses generations. Thus our input events were scaled with 1000/29 as they were in ky and the migrations were multiplied by 10^{-5} to make the correct scaling. We used a more human-readable version for priors instead of the required one to understand the output easily. Of course, we could have used direct generations and the correct amount of migration matrix directly, but it won't be easy to understand. Of course, if we are running the parameter estimation only once, it does not matter. However, it becomes easier when we use recursively to understand if our results are not going awry (especially useful when we are using recursive.sh with snakemake pipeline. See later for more information. You can add your code in the src/SFS/Demography.py, which has to follow some simple rule:

```
import msprime
def demo(params, inds, length=1e6, mutation_rate=1.45e-8, recombination_rate=1e-8, replicates=300):
    your own code
    geno=msprime.simulate(...)
    return geno
```

Creating SFS csv file

After making the correct format for demography, it is time to create the csv file from the simulations.

```
python src/SFS/Run_Prior2SFS.py  OOA --params_file Params.csv --inds 5,5,5 --threads 5 --total_length 1e7 |gzip > OOA.csv.gz
```

- OOA
  This is the name of the demography, which is present in src/SFS/Demography.py. You can use whatever name as the definition and can be accessed from here.
- --params_file Params.csv
  This file is the priors that are produced in the previous step.
- --inds 5,5,5
  The number of individuals per population.
- --threads 5
  The number of threads to be used. As this step is the bottleneck for the whole approach, this is important to use.
- --total_length 1e7
  The total length of the genome has to be simulated. Here we are simulating 10 mbp region. The code will run 1mb region at a time by default. Thus 10mbp

means repeating the 1mbp regions ten times. Because of this approach, I would suggest not to use seeds or use them with caution. If not, it will produce the same SFS again and again. Thus running it ten times will not be any improvement.

This command will create an Out of Africa simulated SFS with parameter file, which then can be directly used as an input for the parameter estimation or SMC method. Undoubtedly, this part is the main bottleneck for the whole approach. Thus this part should be mostly improved if we use an even more, parallelize approach. For example, only using threads will not be enough. In my case, I used cluster to massively parallelize by submitting multiple productions of csv together using snakemake (see later).

Going forward with ABC-DLS (SFS to Posteriors)

With this prior stuff done, finally, we reached a situation where we can use the ABC-DLS method to get out the posterior range.

```
echo -e "OOA.csv.gz\t14" > Model.info
python src/Run_SMC.py All --test_size 5 --tolerance .5 --ssfile examples/Examples.csv --scale b Model.info --decrease 0.95 --increase .0
```

Understandably, we do not expect to reach any level of correctness at all with ten simulations. To have a good level of power, we need to use much more simulations:

```
python src/SFS/Run_Range2UniParameters.py --upper 25e3,2e5,2e5,2e5,1e4,1e4,1e4,80,320,700,50,50,50,50 --lower 5e3,1e4,1e4,1e4,500,500,50(
python src/SFS/Run_Prior2SFS.py  OOA --params_file Params.csv --inds 5,5,5 --threads 5 --total_length 1e6 |gzip > OOA.csv.gz
echo -e "OOA.csv.gz\t14" > Model.info
python src/Run_SMC.py All --ssfile Examples.csv --scale b Model.info --frac 0.16442630347307816 --decrease 0.95 --increase .01 --hardran
```

The frac (fraction) was calculated with the available amount of data for chr22 (for the vcf file), which is 6,081,752. Thus to make it equal with the simulations, we have to multiply 1e6/6081752 or 0.16442630347307816. You will see a decrease (dec) for several parameters. But this ran only once. To use it recursively, we can use the output (Newrange.csv) and use it as input to rerun it till there is no decrease possible. Please check examples/Examples.md for all the information for all the other options.

Snakemake

We added a snakemake pipeline to run those commands automatically. Snakemake pipeline will be beneficial for clusters, where we can run multiple jobs together. Unfortunately, here we can not talk about how to install and implement snakemake pipeline in the cluster. For further information, you have to see snakemake tutorial. The snakemake pipeline takes config.yml as an input, which takes several necessary configuration parameters together inside a yml file. Here we add an example in config.yml file:

```
sc_priors: Run_Range2UniParameters.py
sc_sfs: Run_Prior2SFS.py
sc_abc: ../Run_SMC.py
sfsfile: ../../examples/YRI_FRN_HAN.observed.csv
priors_range: Oldrange.csv
demography: OOA
inds: 5,5,5
threads: 1
jobs: 10
repeats: 2000
total_length: 1000000
test_size: 1000
decrease: 0.95
increase: .01
hardrange_file: Startrange.csv
tolerance: .1
frac: 0.0015455858426908665
```

- sc_priors, sc_sfs and sc_abc are the scripts for running the necessary commands for Run_Range2UniParameters.py, Run_Prior2SFS.py and Run_SMC.py. If you want to run it somewhere else, you can put the full path instead of the relative path.
- sfsfile is the full path where you have your real or observed SFS file in csv format. You can use python src/SFS/Run_VCF2SFS.py to produce such a file. Please see above how to do it. Here we used one real observed data of Yoruba, French and Han Chinese downloaded from High Coverage HGDP data.
- priors_range is the file path for a csv file that has three columns. The first columns are the parameters name, the second column is the lower limit and the third column is the upper limit for every parameter. No header is expected. Example:

```
N_A,5000,25000
N_AF,10000,200000
N_EU,10000,200000
N_AS,10000,200000
N_EU0,500,10000
N_AS0,500,10000
N_B,500,10000
T_EU_AS,15,80
T_B,5,320
T_AF,5,700
m_AF_B,0,50
m_AF_EU,0,50
m_AF_AS,0,50
m_EU_AS,0,50
```

- demography is the name of the demography def, which is saved in the Demography.py file.
- inds is the number of individuals per population.
- threads is the number of threads that would be used per simulation run where we create SFS using msprime using Run_Prior2SFS.py.
- jobs are the number of different jobs you want to run separately. This option will break the Priors.csv into several smaller but independent files that can then run separately in parallel.
- repeats are the number of repeats that are needed to run ABC-DLS SMC. Remember, this does not correspond to the number of the simulation run for this particular loop as we reuse some of the older simulations using Narrowed.csv.
- total_length is the total length of the simulation that we want to run. Remember the total length is divided by equal 1mbp of LD region or chromosome to run it separately.
- test_size is the number of test_size that would be kept for ABC-DLS. Everything else will be used for training.
- decrease is the amount of decrease necessary to regard it as a true improvement. For more information please see examples/Examples.md under SMC.
- increase is the amount that should be added to the posterior ranges if it does not have any improvement or decrease.
- hardrange_file is the starting range file. Important in case you use increase.
- tolerance amount of tolerance that is necessary for ABC analysis.
- frac is the amount of fraction to multiply with observed SFS to be equal to the simulated SFS. You can also write there !!float 1/647

To run the snakemake, you can use just run in this folder:

```
snakemake --jobs 10
```

This snkamake command will run all the necessary commands. It will run parallel ten jobs and will produce a Newrange.csv and Narrowed.csv. But this only one recursion. We need to do multiple recursion to make our posterior range much smaller. To do it, we need to put this code inside a while loop. We should also change the Newrange.csv to Oldrange.csv to run inside a loop until it reaches convergence.

Recursion

The last and final part of this approach is to put it (snakemake pipeline) inside a recursive loop. We can do this in several ways. Here we present a simple shell script approach to do it (recursive.sh).

```
#!/bin/bash
imp=0
touch Narrowed.csv
cp Startrange.csv Oldrange.csv
while [ "$(echo "$imp < 0.95"| bc -l)"  -eq 1 ]
do
        snakemake -q --jobs 6
        imp=$(cut -f4  -d ","  Newrange.csv |sort -n | head -n 1)
        mv Newrange.csv Oldrange.csv
done
mv Oldrange.csv Finalrange.csv
```

To run this code, we have to use:

```
sh recursive.sh
```

This script will automatically run the recursion of the snakemake pipeline inside a while loop. We used Startrange.csv as a starting point of the whole recursion and Oldrange.csv as the starting point of every loop. The snakemake pipeline will submit six jobs in parallel, and when it reaches convergence (which is a minimum of imp > 0.95), it will stop the while loop and save it as Finalrange.csv. This script is a very basic way to do it. Of course, you are free to update it and make it more complex to benefit your analysis.