ABC-DL pipeline on demographic modelling and parameter estimation adapted to ancient DNA properties ABC-DL v1.1

Comments, doubts, questions and bug reports: olga.dolgova@cnag.crg.eu; oscar.lao@cnag.crg.eu

Disclaimer

ABC-DL is a free pipeline written in JAVA. It remains under the rights of the creators Olga Dolgova and Oscar Lao, and can be edited only after their permission.

You can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. Notice that it uses software from third parties, such as fastSimcoal2 and Encog, which may apply their own rules for redistribution and/or modification.

ABC-DL is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You can download a copy of the the GNU General Public License from http://www.gnu.org/licenses/.

Introduction

The paper Clemente et al. "*The genomic history of the Aegean palatial civilizations*" extends a statistical approach based on approximate Bayesian computation (ABC) and deep learning (DL) published by Mondal et al. (2019) to make it suitable for demographic models that include ancient individuals. The original ABC-DL approach was developed by Oscar Lao and deposited in https://github.com/oscarlao/ABC_DL.

The original idea consists in using feedforward neural networks (NN) with several layers using Encog v3.4 (Heaton, 2015) for predicting a demographic model or a parameter from a demographic model from simulated DNA. The input for the NNs is the multidimensional unfolded site frequency spectrum (*jSFS*). This prediction is used in the ABC framework implemented in the R abc (Csilléry et al. 2010) package to estimate the posterior probability of a parameter/model based on the observed data (Figure 1).

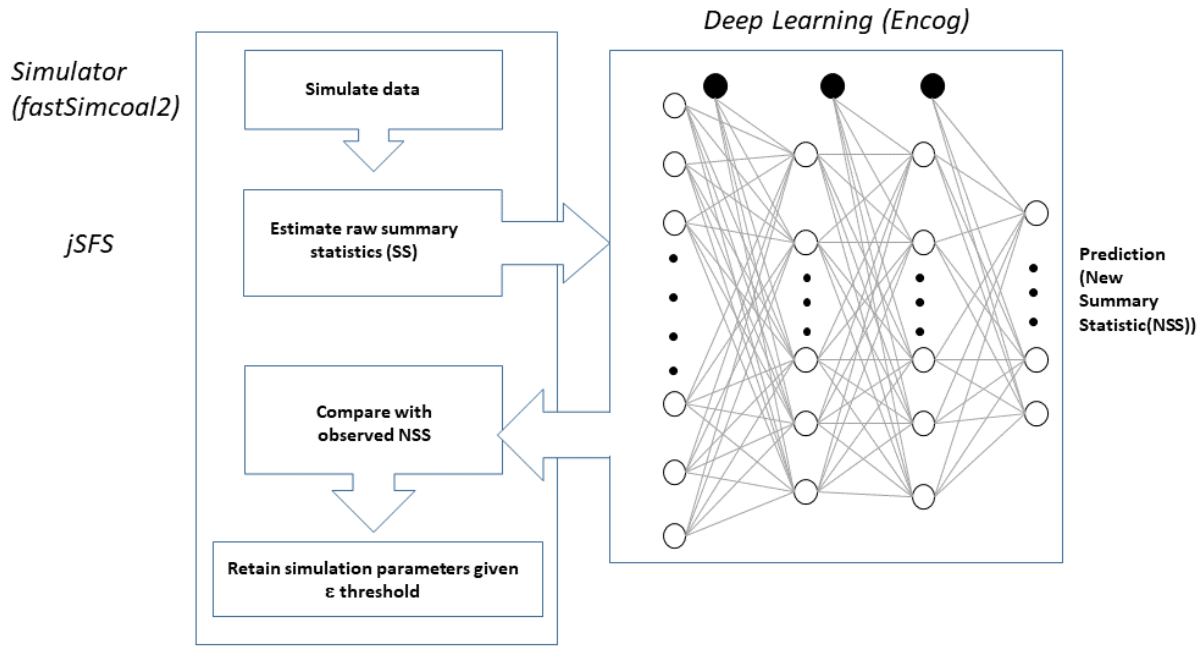

Figure 1. **The ABC-DL pipeline.** Following the ABC paradigm, we generate genomic simulated data from a set of models and parameters, with values ascertained from prior distributions. Out of the genomic data, we generate raw summary statistics (in our case, the *jSFS*). Classical ABC compares the raw summary statistics from the simulated data with the one computed with the observed data, and decides to accept or reject the values used in the parameters as being sampled from the posterior distribution of the parameters given the observed data. Following Wong et al. (2018), we added an intermediate step in which the *jSFS* is used to train Deep Learning neural networks (NNs) to predict the model/parameter that generated the simulation. By definition, this prediction is the most informative summary statistic one can have. Therefore, we use this prediction as summary statistic and compare it with the DL prediction obtained when using the observed data.

The pipeline is written in JAVA. It uses fastSimcoal2 (Excoffier and 2011) to generate the simulations. Here, we included an update of the original FastSimcoalPatch.jar library developed in Mondal et al. (2019) that implements a simple model to emulate some of the basic properties of *aDNA*: modern human contamination, deamination rate and depth of coverage. It takes the *jSFS* output from fastSimcoal2 as "modern data", and the software modifies it by introducing new SNVs due to deamination in invariant positions, removes SNVs that become triallelic due to deamination and, based on the mean depth of coverage, re-codes the genotype of a given individual. If modern human contamination is specified, this extension also allows modifying the *jSFS* to include an excess of shared modern variation with the aDNA sample.

In the proposed framework, **JAVA and R programming skills are prerequisites** for the implementation of a particular model comparison/parameter estimation. A user must be able to write, modify and run JAVA and R scripts. In order to use JAVA packages, you will need to install SDK1.8 and JRI1.8, as well as JAVA editor to import the *src* code. The example project has been created using Netbeans (www.netbeans.org) and uses the external package EncogV3.4 (https://github.com/encog). It also requires fastSimcoal2 software (http://cmpg.unibe.ch/software/fastsimcoal2/) to be available for conducting simulations. **We assume that the reader has the basic knowledge on importing the project in his or her favorite editor and we will not describe how to do that.**

Folder organization

Since ABC requires a large number of time-consuming simulations, we parallelized the simulations per model running independent jobs in a cluster. In practice, we split the simulations in different folders, each containing its own fastSimcoal2 executable, which conditioned the final folder organization of the project.

The structure of folders of an ABC-DL project is:

main_folder
- a. fastSimcoal_0
- b. fastSimcoal_1
- c. …
- d. fastSimcoal_*n*
- e. model
- f. parameter
  - i. model_a
  - ii. model_b
  - iii. …
  - iv. model_k

In *main_folder* we have the observed genomic data in Plink binary format, consisting in three files (.fam, .bim, .bed) (please refer to https://www.cog-genomics.org/plink2/formats). The observed data must contain at least two individuals from each population and an additional individual with name "*Ancestral*" that defines the ancestral allele for each SNP. See the toy data example included in https://github.com/olgadolgova/ABC_DL_aDNA/tree/main/ABC_DL_Example_Project/test_model.
In addition to these files, the software requires a text file called "masked_regions". This file contains the independent regions that are going to be simulated by fastSimcoal2. In principle, these regions should be neutral. In Clemente et al. we assumed that regions out of genes and CGp islands behave neutral, similarly to Mondal et al. (2019). However, other ascertainment scheme of regions can be applied (i.e. Pouyet et al. 2018). We provide tools to generate the "masked_regions" file and an example of the "masked_regions" file in https://github.com/olgadolgova/ABC_DL_aDNA/tree/main/ABC_DL_Example_Project/test_model.
The file with observed data and the list of genomic regions excluding genes and CpG islands must be created before we attempt to run any analysis.

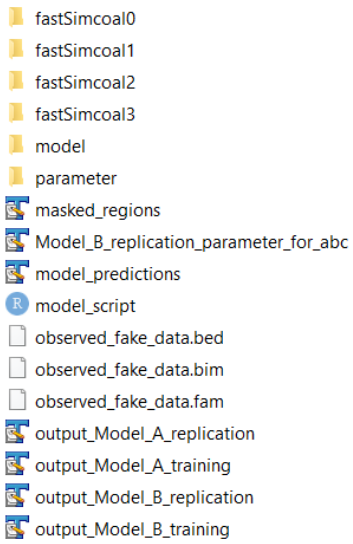

Figure 2. Example of the organization of folders and files required to run the pipeline. Each fastSimcoalX (where X starts at 0) folder contains the executable of the fastSimcoal2 software (i.e. fsc26). Model and parameter folders contain the estimated neural networks. "masked_regions" is a txt file indicating the regions that are going to be simulated. Observed data is stored in binary plink format, involving three files (.fam, .bim, .bed). Out of the simulations, for each model two files will be created: one for training and the other for replication. Each file will refer to a model and the name of the file will be "*output_@_[replication, training]*", where @ is the name of the model as defined in the ABC-DL pipeline.

Each of the fastSimcoal folders contains the executable of fastSimcoal2. The Encog network files for predicting the models are stored in the folder called "*model*". The folder "*parameter*" contains the networks of the parameters of a model.

Creating a new project

The JAVA pipeline is implemented to generate demographic models, setting prior distributions, running fastSimcoal2 with these models, store the genetic information (*jSFS*) of simulated genomes, create artificial NNs (ANNs) using Encog and to create output files that contain the DL predictions for models or parameters from a given model in simulated and observed data. This output is used in a second step in the ABC framework. We have used the R package *abc* (https://cran.r-project.org/web/packages/abc/index.html), but there are many other packages implementing ABC that could be used.

The JAVA project implementing the whole pipeline is called *ABC_DL_Suite*, and requires several *jar* libraries to work. The Encog library can be obtained from https://github.com/encog of Jeff Heaton. The rest of the libraries have been developed by the authors of the paper and are included in the folder "libraries" as *jar* files.

The first step would be to set up the *ABC_DL_Suite* project in your favorite editor. A simple way is to clone the *ABC_DL* repository, opening it in your editor, and setting in *Properties* the *jar* libraries from the "libraries" folder.

Creating a new ABC-DL project will require generating an object from the class *ProjectInformation.* This is a helper class that contains all the information that is required to do the simulations and to store them: the exact location of the main folder, the observed individuals that are going to be used for the training and replication, and the populations used to generate the *jSFS*. Recall that we are going to use ONE individual per population in the training and ONE individual per population in the replication. In principle, the training individuals should not be used for replication, but in some cases, where only one individual is available (which is usual for ancient genomes) the same individual is used for training and replication dividing the corresponding SNPs in two datasets and using different names for these datasets derived from the same individual. In principle, one would like to use ALL the populations for both model and parameter estimation. Nevertheless, the number of cells for SFS, which are used as input in the DL is $3P-2$ for P populations. Therefore, if P is large, the number of input cells in the DL can exceed the capability of the network for identifying useful patterns.

An implementation that uses the *ProjectInformation* would look like this:

```
public class ProjectInformationOfThisImplementation {

/**
* Hard coded information of the test project
* @return
* @throws Exception
*/

public static ProjectInformation getProjectInformation() throws Exception
		{
// Individuals for training
String [] training = {"Ind_0", "Ind_2", "Ind_4","Ind_6"};
// Individuals for replication
String [] replication = {"Ind_1", "Ind_3", "Ind_5","Ind_7"};

// pops to retrieve
int [] pops_to_retrieve = {0,1,2,3};

return new ProjectInformation("observed_data", new ModelsToRunABC_DL(""), "path_to_abc_dl\\test_model\\","fsc26",4, new WINDOWS(),
10000, training, replication, pops_to_retrieve);

		}

}
```

It indicates that all the information of the project is in path_to_abc_dl\test_model\, the executable *fsc26* is stored in four folders fastSimcoal_X, in a windows system, 10,000 replicates will be simulated in each fastSimcoal folder for each model, using the training samples for noise injection and the replication samples for the parameter/model estimation, retrieving all the populations. Therefore, for each model we run 40,000 simulations. Recall that this is a quite small number of simulations for an ABC-DL approach. In a real scenario, we would probably need a much larger number of simulations (≥100,000).

A key point are the models that we are going to run (*ModelsToRunABC_DL*), a list that extends the *Load_Model_Data* class (see below).

Generating a model in the ABC-DL JAVA framework

Before we start thinking in running the ABC-DL, first we have to implement in JAVA the models to test. All classes that implement a model to be run in fastSimcoal2 must extend the abstract class *FastSimcoalModel* and implement:

void defineDemography()

This method defines which are the populations that we are going to use in the fastSimcoal2 simulations and how many chromosomes have been sampled. In the case of usage of any ancient genomes, their properties (deamination rate, mean depth of coverage and proportion of modern human contamination) might be defined also within this method.

void initializeModelParameters()

This method defines the structure of the demographic model (which population splits from which other, effective population changes, etc). The way to define events is, essentially, the same as in fastSimcoal2, going backward in time. This means that a population split in two from the past to the present is modelled in fastSimcoal2 (and hence in the ABC-DL framework) as one of the two populations in the present collapsing into a single one in the past. The population that disappears is called “source” and the one that remains is “sink”. It is very important to keep in mind that, even if in our model the “source” population does not exist anymore going backward in time, it is still possible to call it at fastSimcoal2. In practice, this means that if the “source” population after it has been merged with the “sink” gets migrants from another population, final coalescence will never take place and the run will take forever. The web page of fastSimcoal2 offers some tools to visually check if a generated demographic model shows this type of behavior.

Since in ABC-DL we are interested in generating simulations given a range of possible values for each parameter given our prior knowledge, ABC-DL allows sampling from a distribution for a given event (say, “the time of split between population 1 and population 2 uniformly ranges between 10 and 50 generations”). This distribution is the prior distribution that we use in the ABC framework for ascertaining the values of the parameters to generate the simulations following the priors. The properties of aDNA (deamination rate, mean depth of coverage and proportion of modern human contamination) may be also defined as parameters to estimate establishing appropriate prior distributions. Nevertheless, ABC-DL also allows setting a single value when the parameter is fixed.

Next, we explain the pipeline to generate the model of Figure 3.

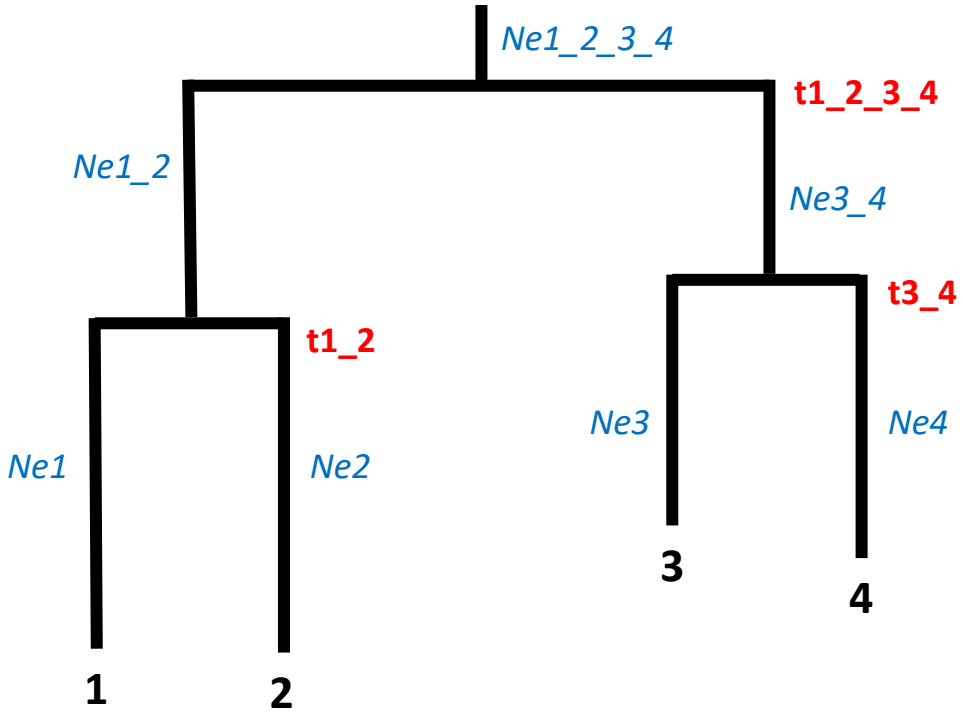

Figure 3. Model A. Ne_x = effective population size of population x. t_{x_y} = time of split between population x and y.

This model considers 10 parameters, including effective population sizes (Ne_x) and time of splits (t_{x_y}). Furthermore, population 3 and population 4 are sampled at archaic times.

First, we create a new class that extends *FastSimcoalModel* abstract class:

```
public class Model_A extends FastSimcoalModel {
```

Next, we instantiate the *defineDemography* method from the super class:

```
@Override
protected void defineDemography() throws ParameterException {
```

We have to provide a name for each of the populations. This name must be unique for each population and it is case sensitive. Every time we need to call one of the populations, we do it using the proposed name.

```
// Names of the populations to be used in fastSimcoal2
String [] names = {"Pop1", "Pop2", "Pop3", "Pop4"};
demography = new Demography(names);
```

Demography is a class that stores the demographic information of the populations, including how many chromosomes have been sampled, and the demographic history (in terms of Events). If the samples are from present times, we only have to call:

```
demography.addSample(names[0], 2);
demography.addSample(names[1], 2);
```

Where *names[0]* refers to the name of the first Population ("Pop1") and 2 refers to the number of chromosomes that we sample. In principle, ABC-DL assumes one individual (two chromosomes) per population.

If the samples are from ancient times, then we call:

```
demography.addSampleWithTime(names[2], 1800, 2); //Archaic population 1 is sampled 1800 generations ago
demography.addSampleWithTime(names[3], 1414, 2); //Archaic population 2 is sampled 1414 generations ago
}
```

Here, *addSampleWithTime* has an extra argument to specify the sampling time as second parameter. The third parameter corresponds to the number of chromosomes that we sample.

To define the rate of deamination of each ancient sample, first of all we define the expected proportion of A, C, T, G nucleotides in a human genome, which is not affected by deamination:

```
this.percentage_ACTG_in_genome[0] = 0.3; // proportion of A in human genome
this.percentage_ACTG_in_genome[1] = 0.2; // proportion of C in human genome
this.percentage_ACTG_in_genome[2] = 0.3; // proportion of T in human genome
this.percentage_ACTG_in_genome[3] = 0.2; // proportion of G in human genome
```

Then, we add observed deamination rate (which can be previously calculated using, for example, ANGSD tool (Korneliussen et al., 2014)) to each of the ancient genomes:

```
demography.setPopulationIIsAncient(names[2], new Value(0.0105), new Value(0.0105), new Value(4));
demography.setPopulationIIsAncient(names[3], new Value(0.0088), new Value(0.0088), new Value(5));
```

The first and second parameter correspond to the proportion of G->A and C->T, respectively, which are usually almost equal. The last value corresponds to the mean depth of coverage of an ancient sample.

To designate the source and proportion of modern human contamination calculated previously, we call:

```
demography.addContamination(names[1], names[2], new Value(0.01));
demography.addContamination(names[1], names[3], new Value(0.0058));
```

Here, the first argument is the name of a modern sample supposed to be the source of contamination, second argument is the name of an ancient sample, third argument is the proportion of contamination. This implies that the demography of the modern contaminant, as well as its position in the topology of the demographic model, must be fully specified.

Next, we need to specify the demographic model in the *initializeModelParameters()* method using a fastSimcoal2-like language. The common way to define events and initialize the effective population sizes of the populations is to specify the names of the source and receptor populations. For a detailed definition of how fastSimcoal2 specifies the events, please check the fastSimcoal2 manual.

In the current example, we first define the effective population sizes of the four populations:

```
ParameterValue Ne1, Ne2, Ne3, Ne4, Ne1_2, Ne3_4, Ne1_2_3_4;
```

Each effective size is coded in a *ParameterValue* object. ABC-DL accepts three types of objects defining a value, all of them inheriting from the *Parameter* class. Fixed values can be defined using the *Value* class. Values depending on a sampling from a probability distribution use *ParameterValue*, and parameters that depend on other parameters (for example, the time of split of two populations might depend on the fact that backward in time there was first an event of admixture between them) use the class *ParameterValueScalable*. We will see each of them in the example.

The *ParameterValue* object contains a probability function that indicates the prior of the parameter we want to estimate. For example, for the first population, we define an effective population size with a uniform prior distribution that ranges between 1000 and 5000 chromosomes.

```
Ne1 = new ParameterValue("Ne1", new DistributionUniformFromValue(new Value(1000), new Value(5000)));
```

Each *ParameterValue* requires a String name and a *Distribution* type object.
DistributionUniformFromValue extends the *Distribution* class and it takes two parameters. These two parameters are fix values.

Since this is a parameter we want to estimate, we include it in the parameters list:

```
parameters.add(Ne1);
```

Any parameter that has to be estimated must be included in the parameters list.

Since this is the current population size of population 1 (Pop1), we need to update the demography object with the effective population size object Ne1 in FastSimcoal2:

```
demography.addEffectivePopulationSize("Pop1", Ne1);
```

The program, once called, will generate a FastSimcoal2 demography input file sampling one value from the prior distribution and this value will be stored in the simulation output file.

We need to do the same procedure for the other populations, even if they are not sampled:

```
// The effective population size of pop 2 can range A PRIORI between 500 and 1000

Ne2 = new ParameterValue("Ne2", new DistributionUniformFromValue(new Value(500), new Value(1000)));
parameters.add(Ne2);
demography.addEffectivePopulationSize("Pop2", Ne2);

// The effective population size of pop 3 can range between 10000 and 20000

Ne3 = new ParameterValue("Ne3", new DistributionUniformFromValue(new Value(10000), new Value(20000)));
parameters.add(Ne3);
demography.addEffectivePopulationSize("Pop3", Ne3);

// The effective population size of pop 4 can range between 5000 and 10000

Ne4 = new ParameterValue("Ne4", new DistributionUniformFromValue(new Value(5000), new Value(10000)));
parameters.add(Ne4);
demography.addEffectivePopulationSize("Pop4", Ne4);
```

Not defining the effective population sizes of the simulated populations will produce a nice crash and error message saying that there are undefined parameters in the model.

Now we need to add the events. An event corresponds to an interaction between populations at a given time. In our case, every time there is an event, the effective population size of the involved populations changes. Therefore, each event will have TWO parameters associated: the time when it occurs, and the new effective population size. First, we define the parameters corresponding to time.

```
// TIME EVENTS

ParameterValue t1_2, t3_4;
```

Recall that we are using fastSimcoal2 language, which works backward in time. Hence, populations that split in forward, merge in backward. Let us consider the first population merging backward in time between population 1 and 2:

```
// EVENTS
```

Similar to when defining the effective population size, we define the range of values that the time of split (in forward) or merge (in backward) can take.

```
// Event Split pop1 from pop2. It ranges between 500 generations and 1500 generations

t1_2 = new ParameterValue("tSplitPop1_Pop2", new DistributionUniformFromValue(new Value(500), new Value(1500)));
```

We include it in the parameters list:

```
parameters.add(t1_2);
```

We do the same for the new effective population size of the ancestral population that split in population 1 and 2 forward in time.

```
// The effective population size of this merged population can be between 100 and 200

        Ne1_2 = new ParameterValue("Ne1_2", new DistributionUniformFromValue(new Value(100), new Value(200)));
         parameters.add(Ne1_2);
```

The event is then specified by means of an object from the *EventParameter* class, which uses fastSimcoal2-like language. We will need to indicate the time when it occurs (the object t1_2 that we had defined), the population that merges to another population (and stops existing) which is “Pop1” in this case, the other population that receives the migrants from the first population (“Pop2”), which in this case corresponds to all the migrants (hence, the value 1.0), the previous effective population size of the population that receives the migrants (Ne2), and the new effective population size (Ne1_2). The last two parameters correspond to the new growth rate, which in our case is 0 (we are considering a constant population size) and the migration matrix. After generating the *EventParameter*, we add it to the demography.

```
// In fastSimcoal2, a split event forward is defined as an event backward where one of the two populations merges all its migrants with the other population.

        EventParameter etSplit1_2 = new EventParameter(t1_2, demography.getPosition("Pop1"), demography.getPosition("Pop2"), new Value(1.0), Ne2, Ne1_2, new Value(0), 1);
        demography.add_event(etSplit1_2);
```

Recall that we are using the fastSimcoal2 format. In a split (forward in time) or merge (backward in time), any of the two populations can be the receptor/source. However, once we have decided that one of the two populations is the receptor, this has to be kept during the remaining of the tree topology.

Similarly, for population 3 and 4 we define the split event as population 3 merging with population 4:

```
t3_4 = new ParameterValue("tSplitPop3_Pop4", new DistributionUniformFromValue(new Value(2000), new Value(4000)));
parameters.add(t3_4);

Ne3_4 = new ParameterValue("Ne3_4", new DistributionUniformFromValue(new Value(100), new Value(200)));
parameters.add(Ne3_4);

EventParameter etSplit3_4 = new EventParameter(t3_4, demography.getPosition("Pop3"), demography.getPosition("Pop4"), new Value(1.0),
Ne4, Ne3_4, new Value(0), 1);
demography.add_event(etSplit3_4);
```

When we merge the ancestral population 1_2 with 3_4, we have to take into account that the time of split must be older than the split of population 3 and 4. Here we use another type of distribution:

```
// TIME EVENTS THAT DEPEND ON ANOTHER EVENT.

// In this case, the time of split of the ancestors of (pop1,2) and (pop3,4) cannot be younger than the time of split of pop3,4

ParameterValueScalable t1_2_3_4;

// Event split pop1_2 from pop3_4

t1_2_3_4 = new ParameterValueScalable("tSplitPop1_Pop2_Pop3_Pop4", new DistributionUniformFromParameterValue(t3_4, new
Value(6000)));
t1_2_3_4.setScalable(t3_4);
```

The object from the class *DistributionUniformFromParameterValue* can take as range the value from another distribution (in this case, the range defined by the distribution [2000, 4000] generations from t3_4). Thus, when performing the simulation, first a value within the range of [2000, 4000] is sampled and assigned to t3_4 and then a value is sampled from [t3_4, 4000] and assigned to t1_2_3_4. Continuing with the implementation, we add this parameter and the Ne of the ancestral population to the parameters list, and create the event:

```
parameters.add(t1_2_3_4);

Ne1_2_3_4 = new ParameterValue("Ne1_2_3_4", new DistributionUniformFromValue(new Value(100), new Value(200)));
parameters.add(Ne1_2_3_4);
```

Recall that after the first merging, population 1 does not exist anymore. After the second merging, population 3 is also out. Population 1_2 is, in fact, population 2. Similarly, population 3_4 is population 4. Here we merge the lineages of population 2 into population 4 and change the size of population 4 (which was Ne3_4) to the new effective population size.

```
EventParameter etSplit1_2_3_4 = new EventParameter(t1_2_3_4, demography.getPosition("Pop2"), demography.getPosition("Pop4"), new
Value(1.0), Ne3_4, Ne1_2_3_4, new Value(0), 1);
demography.add_event(etSplit1_2_3_4);

        }
```

The characteristics typical for aDNA, such as proportions of substitutions due to deamination and modern human contamination, can be also included in the model as parameters to estimate, defining the corresponding prior distributions instead of fixed observed values.

To define the priors of proportions of substitutions due to deamination ranging from 0% to 2% in the sample Pop3 we call:

```
// Ancient DNA

ParameterValue deamination_Pop3_CtoT = new ParameterValue("deamination_Pop3_CtoT", new DistributionUniformFromValue(new Value(0.0), new Value(0.02)));
parameters.add(deamination_ Pop3_CtoT);

ParameterValue deamination_Pop3_GtoA = new ParameterValue("deamination_Pop3_GtoA", new DistributionUniformFromValue(new Value(0.0), new Value(0.02)));
parameters.add(deamination_Pop3_GtoA);

demography.setPopulationIIsAncient(names[2], deamination_Pop3_CtoT, deamination_Pop3_GtoA, new Value(4));
```

To define the priors of modern human contamination ranging from 0 to 25% for the same sample we call:

```
ParameterValue contamination_Pop3 = new ParameterValue("contamination_Pop3", new DistributionUniformFromValue(new Value(0.0), new Value(0.25)));
parameters.add(contamination_Pop3);
demography.addContamination("Pop2", " Pop3", contamination_Pop3);
```

We would have finished the model implementation for this model.

Now we can implement another model, now including an introgression event between population 3 and population 4:

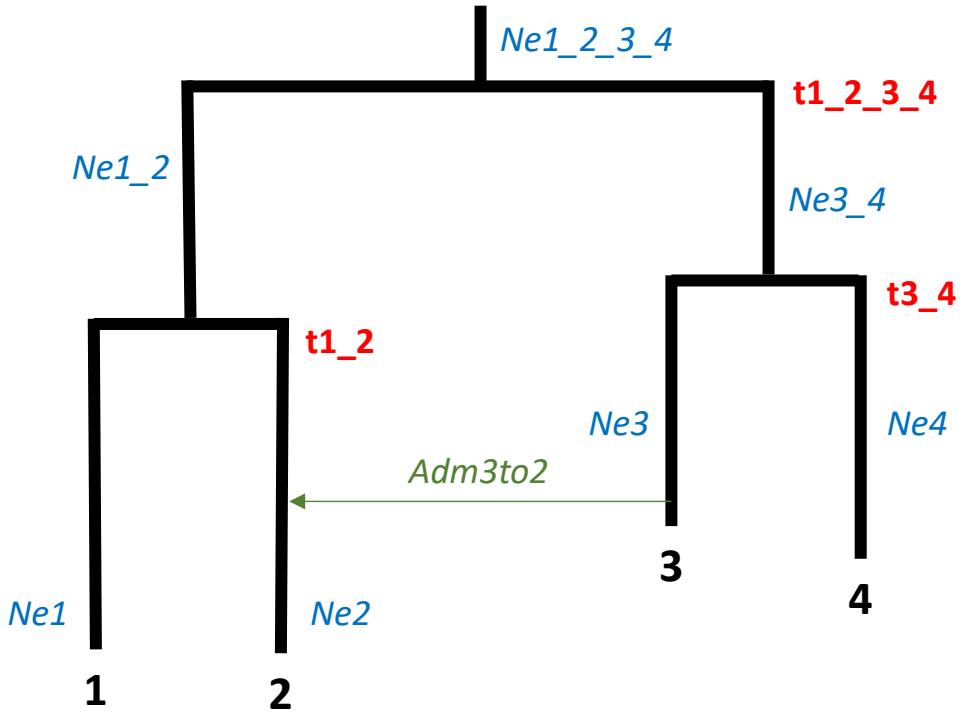

Figure 4. Model B. This model is the same as model A, but includes an introgression event between population 3 and 2.

The only difference in the code between this model and model A is the event of introgression:

```
// Event of introgression. Population 2 sends backward in time migrants to Population 3 (see FastSimcoal2 for details)

// Event Archaic introgression of population 3 in 2 between 200 and 400 generations ago

tintrogression3to2 = new ParameterValue("tIntrogressionPop3_to_Pop2", new DistributionUniformFromValue(new Value(200), new
Value(400)));
parameters.add(tintrogression3to2);

// Introgression ranges between 1% and 20%

ParameterValue introgression = new ParameterValue("IntrogressionPop3_to_Pop2", new DistributionUniformFromValue(new Value(0.01), new
Value(0.2)));
parameters.add(introgression);

// In fastSimcoal2, a split event forward is defined as an event backward where one of the two populations merges all its migrants with the
other population.

EventParameter eIntrogression3_to_2 = new EventParameter(tintrogression3to2, demography.getPosition("Pop2"),
demography.getPosition("Pop3"), introgression, new Value(1.0), new Value(1.0), new Value(0), 1);
demography.add_event(eIntrogression3_to_2);
```

For the introgression event, we define two new parameters (the time of the introgression and the amount of introgression). The code of this event considers that population 2 (source) sends to population 3 (sink) a number of variants at the time of introgression counted in backward manner (check FastSimcoal2

manual). From a forward point of view, population 3 sends migrants to population 2. All the other parameters of population 3 do not change (hence, the value 1 for all the other fields).

We will use these two models to perform an ABC-DL analysis. To specify this in the project we create a class that extends *Load_Data_Model* and implements *defineModels()* method:

```
public class ModelsToRunABC_DL extends Load_Model_Data{

public ModelsToRunABC_DL(String folder) throws ParameterException{

super(folder);

}

@Override

protected void defineModels() throws ParameterException {

bmodel = new FastSimcoalModel[2];

bmodel[0] = new Model_A();

bmodel[1] = new Model_B();

            }
}
```

In order to consider a more realistic case in our ABC-DL framework, we will use data from a model that is similar to one of the two previously considered models for model comparison:

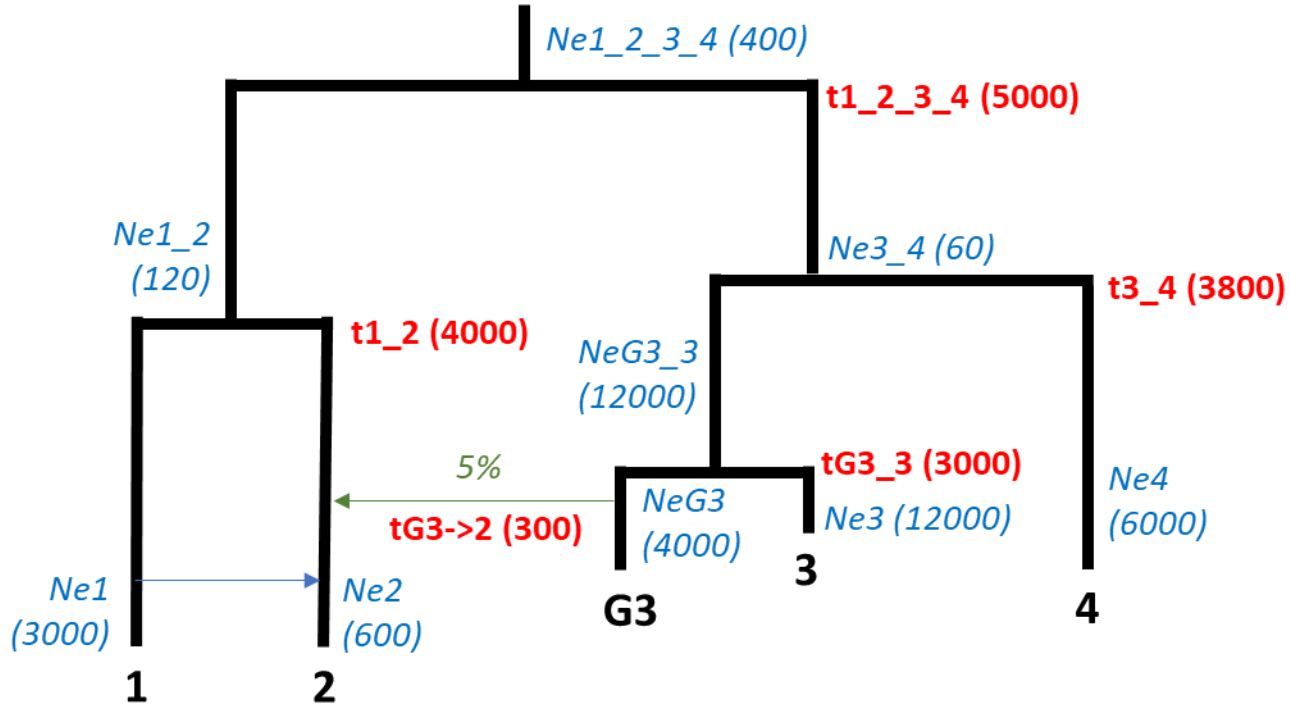

Figure 3. ModelR. The model that generated the real data. We assume that this model is unknown, but related to a certain extent to the models that we are going to compare (Model A and Model B).

The code for this model is similar to model B, but it includes an additional *unsampled* ghost population (G3), and gene flow from population 1 to population 2. Furthermore, since it is the observed data, all the values of the different parameters are fixed.

```
public class Model_R extends FastSimcoalModel {

@Override

protected void defineDemography() throws ParameterException {

// Names of the populations to be used in fastSimcoal2

String[] names = {"Pop1", "Pop2", "Pop3", "Pop4", "G3"};

	demography = new Demography(names);

// How many chromosomes are sampled from each population? In principle, only two chromosomes (one individual)

// Modern populations

for (int pop = 0; pop < 2; pop++) {

	demography.addSample(names[pop], 2);

		}

// Ancient populations

	this.percentage_ACTG_in_genome[0] = 0.3;
	this.percentage_ACTG_in_genome[1] = 0.2;
	this.percentage_ACTG_in_genome[2] = 0.3;
	this.percentage_ACTG_in_genome[3] = 0.2;

	demography.addSampleWithTime("Pop3", 1800, 2); //Archaic population 1 is sampled 1800 generations ago
	demography.setPopulationIIsAncient("Pop3", new Value(0.0088), new Value(0.0088), new Value(5));

	demography.addSampleWithTime("Pop4", 1414, 2); //Archaic population 2 is sampled 1414 generations ago
	demography.setPopulationIIsAncient("Pop4", new Value(0.0105), new Value(0.0105), new Value(4));

	demography.addSample("G3",0);

//	Modern human contamination
	demography.addContamination("Pop2", "Pop3", new Value(0.01));
	demography.addContamination("Pop2", "Pop4", new Value(0.0058));


}


@Override

protected void initializeModelParameters() throws ParameterException {

// Migrations (see below for clarification of how to set up migrations)

MigrationMatrix migrationMatrix = demography.getMigrationMatrix();

// MIGRATION RATES

ParameterValue migrationPop1_to_Pop2 = new ParameterValue("migration_Pop1_to_Pop2", new DistributionUniformFromValue(new
Value(0.0), new Value(5 * Math.pow(10, -4))));
parameters.add(migrationPop1_to_Pop2);

migrationMatrix.addMigration("Pop1", "Pop2", migrationPop1_to_Pop2);
```

```
// Demography

demography.addEffectivePopulationSize("Pop1", new Value(3000));
demography.addEffectivePopulationSize("Pop2", new Value(600));
demography.addEffectivePopulationSize("Pop3", new Value(12000));
demography.addEffectivePopulationSize("Pop4", new Value(6000));
demography.addEffectivePopulationSize("G3", new Value(4000));

// Introgression G3 -> Pop2

EventParameter eIntrogression3_to_2 = new EventParameter(new Value(300), demography.getPosition("Pop2"),
demography.getPosition("G3"), new Value(0.05), new Value(1.0), new Value(1.0), new Value(0), 1);
demography.add_event(eIntrogression3_to_2);

// Split G3 with Pop3

EventParameter eSplitG3_3 = new EventParameter(new Value(3000), demography.getPosition("G3"), demography.getPosition("Pop3"), new
Value(1), new Value(1.0), new Value(1.0), new Value(0), 1);
demography.add_event(eSplitG3_3);

// In fastSimcoal2, a split event forward is defined as an event backward where one of the two populations merges all its migrants with the
other population.

EventParameter etSplit1_2 = new EventParameter(new Value(750), demography.getPosition("Pop1"), demography.getPosition("Pop2"), new
Value(1.0), new Value(600), new Value(120), new Value(0), 1);
demography.add_event(etSplit1_2);

// Event Split pop3 from pop4.

// In fastSimcoal2, a split event forward is defined as an event backward where one of the two populations merges all its migrants with the
other population.

EventParameter etSplit3_4 = new EventParameter(new Value(3800), demography.getPosition("Pop3"), demography.getPosition("Pop4"), new
Value(1.0), new Value(6000), new Value(60), new Value(0), 1);
demography.add_event(etSplit3_4);

// Event split pop1_2 from pop3_4

EventParameter etSplit1_2_3_4 = new EventParameter(new Value(5000), demography.getPosition("Pop2"), demography.getPosition("Pop4"),
new Value(1.0), new Value(60), new Value(400), new Value(0), 1);
demography.add_event(etSplit1_2_3_4);

        }
}
```

This demographic model includes migrations, which are specified in the FastSimcoal2 format:

```
MigrationMatrix migrationMatrix = demography.getMigrationMatrix();

// MIGRATION RATES

ParameterValue migrationPop1_to_Pop2 = new ParameterValue("migration_Pop1_to_Pop2", new DistributionUniformFromValue(new
Value(0.0), new Value(5 * Math.pow(10, -4))));
parameters.add(migrationPop1_to_Pop2);
migrationMatrix.addMigration("Pop1", "Pop2", migrationPop1_to_Pop2);
```

At each level of the tree topology, a migration matrix between pairs of populations is generated. For this, the source and receptor populations and the migration rates (a number of migrants per generation) between both should be specified. Whenever two populations exchanging migrants merge backward in time, the migration matrix is updated and the migration rate between both populations is set to 0.

Running simulations

Simulations run in the folders fastSimcoal_x (where x ranges between 0 and the total number of parallel executions of FastSimcoal2) that we have specified in the *ProjectInformation* method.

In order to run a simulation, the fragments we want to simulate should be specified in the file "masked_regions.txt" within the working folder, which must be generated BEFORE running any simulation.

The format of the fragments that are we are going to simulate is:

1 1 10000000 : 1 10000000

2 1 10000000 : 1 10000000

In this example, we are going to simulate two regions, each of 10Mb (notice that a REAL project will use the whole genome). We consider that the whole region is callable. If that was not the case, we could, for example, specify which are the callable fragments within each region:

1 1 10000000 : 1 5000000, 6000000 9000000

2 1 10000000 : 1 10000000

In the example above region 1, which comprises 10Mb, has two callable fragments, one starting at position 1 until 5Mb and another starting at 6Mb until 9Mb. The rest is not callable.

Once this file is created, we can run the simulations for each model, calling the fastSimcoal_x folder:

```
public static void main(String [] args) throws Exception

{

// Run the simulations.

GenerateSimulationsSFS.runSimulations(ProjectInformationOfThisImplementation.getProjectInformation(), Integer.parseInt(args[0]));

}
```

The method *runSimulations* of class *GenerateSimulationsSFS* will call fastSimcoal2 using the information of the *ProjectInformation* and a number, which is decomposed in two numbers using the total number of models:

fastSimcoal folder that is going to be used = *args[0]/ number_of_models*;

model that is going to be generated = *args[0] % number_of_models*.

For example, if we have two models and four fastSimcoal folders in our working directory, then *args[0]* can range between 0 and 7. Coded in this way, we can use a job array in our cluster to call all the models at all the fastSimcoal folders at the same time.

Imagine we say *args[0] =* "0" using the two models we have generated previously. That is, we are going to generate simulations of Model_A using the fastSimcoal_0 folder.

Upon execution of the main method, a file under the name output_Model_A.txt with simulated data is created in the fastSimcoal_0 folder. The first row of this file contains the names of the *k* parameters of Model_A that we defined when specifying each parameter in the model script. The next row is a result of a simulation. The first *k* columns correspond to the simulated values. The next $3P-2$ columns for P populations correspond to the *jSFS*, computed by FastSimcoal2:

Ne1 Ne2 Ne3 Ne4 tSplitPop1_Pop2 Ne1_2 tSplitPop3_Pop4 Ne3_4 tSplitPop1_Pop2_Pop3_Pop4 Ne1_2_3_4

2872 831 16205 8469 828 132 3436 169 5473 111 1351					*93*	*1219*	*30*	*27*	*44*	*22*	*729*
454	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*121*	*0*	*0*
0	*0*	*0*	*0*	*0*	*0*	*617*	*0*	*0*	*0*	*0*	*0*
0	*0*	*0*	*12*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
27	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*43*	*0*	*0*
0	*0*	*0*	*0*	*0*	*0*	*11*	*0*	*0*	*0*	*0*	*0*
0	*0*	*0*	*1680*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	

Since we specified in the *ProjectInformation* that for each model we will generate 10,000 simulations in each of the four fastSimcoal folders, after running the script for all the range of possible values (0 to 7), we will have 40,000 simulations of each model.

Next, we need to generate a single simulation output file for each model, calling:

```
GenerateASingleFileWithSimulationsOfAModel.generateASingleFileOfModel(ProjectInformationOfThisImplementation.getProjectInformation(), model,0.5);
```

where *model* is the numbering of the model (in our example, 0 or 1), and 0.5 defines that a half of the number of simulations will be used for training of the NN and another half will be used for the replication in ABC analysis. This will generate two text output files containing the output from the simulations (parameters and unfolded SFS) for each model in the main folder under the names:

output_*namemodel*_training.txt

output_*namemodel*_replication.txt

Generating neural networks for model prediction

Once we have generated simulated data for the different models, we need to generate NNs that predict a model given *jSFS*. When doing this, we need to include *jSFS* from the real data as noise injection that will be used for the training but not for the model prediction in the observed data.

The training samples have been previously defined in the *ProjectInformation* object. The name must correspond to the one that is in the .fam file. Since the observed data contains two samples from each population plus a sample called "Ancestral" that includes ancestral alleles, we can use one sample from each population for noise injection during the training of the networks.

To generate a NN, which predicts the model under which the *jSFS* was generated, we call:

```
GenerateModelComparison.generateModelComparison(0.025, 0, ProjectInformationOfThisImplementation.getProjectInformation(), 20, 6.0);
```

The first number corresponds to the error threshold reaching that the training of the NN should stop. The second number corresponds to the id of the network. In principle, for a given prediction it is suggested to run more than one NN (i.e. 10) and collapse the information from each one into a single prediction at the ABC step. The third argument is the project information that indicates where to store the NNs. The fourth argument is the number of neurons at each intermediate layer. The last argument corresponds to the maximum amount of time in hours we allow for the training of the network (in this case, 6.0 represents six hours. 6.5 would represent six hours and a half).

A typical run will look like:

```
Error: 0.47008417344032816
Error: 0.4700848230612131
…
Error: 0.02503455888012451
Error: 0.025065266326679745
Error: 0.02496877693137553
```

Where the error decreases until reaching the threshold. If this is not achieved within the amount of hours , the network training will stop as it is. Therefore, after training the NN, it is desirable to check the evolution of the error and which is the final error that the NN achieves. In our experience, large errors >0.05 produce meaningless results.

If everything worked fine, a NN file is created in the folder model:

```
ABC_Models_comparison_10000_ThreeLayers20_0
```

Where 20 indicates the number of intermediate neurons and 0 the id of the network. We can run different networks at the same time, getting a set of independent classifications for a given SFS dataset, which can then be combined into a single prediction.

Estimating the posterior probability of each model given the observed data and the generated NN

Let's assume we train 10 independent NN. In the model folder we will have 10 files, each constituting a NN. Now, we want to make the model prediction for the observed data, which was not used for the noise injection during the training, using the simulations that were not used for training the NNs and were stored in the "output_x_replication.txt" files.

In order to generate the prediction output, we call:

```
ComputeModelPredictionInReplicationSimulations.predictModelPosteriorReplicationSimulations(ProjectInformationOfThisImplementation.get
ProjectInformation());
```

This call creates a file in the working directory called *model_predictions.txt*. The first row corresponds to the predictions from the observed data. The next rows correspond to the prediction of one of the replication simulations for each model. Therefore, the total number of rows corresponds to the number of simulations that we have run for replication for each of the models + 1, corresponding to the predictions in the observed data.

The first column corresponds to the name of the model that generated the data ("observed" in the case of observed data). The next columns correspond to the NN predictions. Given K models, the first K columns correspond to the prediction of the first NN. The next K columns correspond to the second NN and so on.

In the case of the working example, the output of the two first rows would look like this:

```
observed 3.1180748445061 09E-7 0.9999996881925155 … 1.0781230284382564E-76 1.0
Model_A 0.8857360522120838 0.11426394778791622 ... 0.7867721150684731 0.21322788493152697
…
Model_B 3.128568500060543E-6 0.9999968714315 … 1 1.4942891071544577E-11 0.9999999999850572
```

The observed data has a probability of ~0 of being Model_A and ~1 of being Model_B for the first NN.

For the first simulation of Model_A, the probability of being generated under the model A is 0.88 for the first NN.

Using this output as observed summary statistics, we can conduct ABC approach to compute the posterior probability of each model given the observed data. In our implementation, we used *abc* package (Csilléry et al., 2010) in R. The script to run the ABC model selection in R is:

```
rm(list=ls());

if("abc" %in% rownames(installed.packages()) == FALSE)

{

stop("abc package is required");

}

library("abc"); # load the abc package

abc_folder <- "path_to_abc_dl"; # CHANGE TO THE FOLDER WHERE YOU RUN THE SIMULATIONS

# now, for each of the neural network replicates, compute the mean distance. Return the sum
```

```
compute.mean.nn <- function(simulated.ss, nen, n.models)
{
simulated.ss.by.nn <- matrix(nrow=nrow(simulated.ss),ncol=n.models, rep(0,nrow(simulated.ss)*n.models));
for(nn in 1:nen)
{
sequence <- seq(from=(1+n.models*(nn-1)),to = (n.models+n.models*(nn-1)),by=1);
simulated.ss.by.nn <- simulated.ss.by.nn + simulated.ss[,sequence];
}
return(simulated.ss.by.nn/nen);
}
# data with the model predictions from the DL
data.t <- read.table(file=paste(abc_folder,"model_predictions.txt",sep="\\"),header=F);
# first row is the observed data
observed.ss <- data.t[which(data.t[,1]=="observed"),2:ncol(data.t)]
# summary statistics of the simulated data. Remember. If we have K models, each K columns correspond to a NN prediction.
simulated.ss <- as.matrix(data.t[-which(data.t[,1]=="observed"),2:ncol(data.t)]);
# names of the models
models.by.row <- data.t[-which(data.t[,1]=="observed"),1];
# levels of the models
model.names <- levels(as.factor(as.character(models.by.row)));
# set each model as an integer
models <- rep(-1,nrow(simulated.ss));
for(m in 1:length(model.names)) {
models[models.by.row==model.names[m]] <- m;
}
# compute the mean predicted value over all the NN for each possible model. First parameter is the matrix of simulations and predictions.
Second parameter is the number of neural networks that we have run. The third parameter is the number of models.
mean.ss <- compute.mean.nn(simulated.ss, ncol(simulated.ss)/ length(model.names), length(model.names));
# do the same for the observed data.
mean.observed.ss <- compute.mean.nn(observed.ss, ncol(simulated.ss)/ length(model.names), length(model.names));
# use postpr to generate the posterior distribution of the data. Check the function in abc package.
res <- postpr(mean.observed.ss, models, mean.ss, tol=1000/nrow(simulated.ss),method="mnlogistic");
```

When we run this code, we see that there is no need to run the *mnlogistic* algorithm for weighting the posterior probability of the models, because there is only one model accepted: Model_B. This is not completely unexpected, since the real model that generated the data is, in fact, a modified version of Model_B.

Generating the Neural Networks for the Parameters of Model B

Now that we have identified model B as the one showing the highest posterior probability, the next step would be to compute the posterior probability of each of the parameters of this model. We can run different NN replicates for each parameter, similar to what we did for the model comparison. In order to generate a replicate of the NN of a parameter from a model, we call:

```
TrainNetworkOfParameter.trainNetworkOfParameterFromModel(ProjectInformationOfThisImplementation.getProjectInformation(), 1, 0, 11, 200, 0.025, 0.5);
```

We need to pass the project information, the model we want to consider (following the example, we want to use Model_B, which in the list of models is the second or 1), the neural network replicate (0 in this example), the parameter we want to run (11, which corresponds to Ne1_2_3_4 in Model_B), the number of intermediate neurons (200), the error threshold to stop (0.025) and the maximum amount of time in hours we allow the NN to run (in the example, 0.5 corresponds to 30 minutes).

As a result of this call, a file with a trained Encog network is generated:

…\Parameter\Model_B\ name_parameter_replicate.network

Once we have generated several replicates of a NN for each parameter, we make the prediction of each parameter in the replication dataset calling:

```
PredictParameters_DL.predictABCParameters(ProjectInformationOfThisImplementation.getProjectInformation(), 1, 1);
```

The first argument is the project information. The second argument is the model we want to use (Model_B in our case). The third argument is the number of replicates we have done of each parameter (only one NN).

After running this line of code, we get an output file in the working directory called Model_B_replication_parameter_for_abc.txt. The format of this file is:

```
Parameter1 | prediction_parameter1_replicate_0 | prediction_parameter1_replicate_1 | …
```

The first row corresponds to the parameter names.

```
Ne1 Ne1_p_0 Ne2 Ne2_p_0 Ne3 Ne3_p_0 Ne4 Ne4_p_0 tIntrogressionPop3_to_Pop2 tIntrogressionPop3_to_Pop2_p_0 IntrogressionPop3_to_Pop2 IntrogressionPop3_to_Pop2_p_0 tSplitPop1_Pop2 tSplitPop1_Pop2_p_0 Ne1_2 Ne1_2_p_0 tSplitPop3_Pop4 tSplitPop3_Pop4_p_0 Ne3_4 Ne3_4_p_0 tSplitPop1_Pop2_Pop3_Pop4 tSplitPop1_Pop2_Pop3_Pop4_p_0 Ne1_2_3_4 Ne1_2_3_4_p_0
```

The next row corresponds to the predicted values of the parameters using the observed data. For each parameter, the first column has value NA, because we do not know which is the value of the parameter.

NA 0.4490767571974948 NA 0.5400730483999845 NA 0.884941037741885 NA 0.536380613137984 NA 0.35803623204054746 NA 0.43550283675165247 NA 0.3467322551912092 NA 0.21781228556682514 NA 0.4798209738666576 NA 0.16046876815861877 NA 0.47164240649401107 NA 0.5417149230059182

The next rows correspond to the parameter values of the simulated data and the predicted values. In our example, an output could look like this:

2232.0 0.18244152810110764 722.0 0.7609947589815826 15547.0 0.16522245352195236 9158.0 0.5714252110619659 318.0 0.4461353731306063 0.07908481221757381 0.2068236769319749 501.0 0.3351961069491004 170.0 0.6892949602832428 2191.0 0.3861632979951155 174.0 0.6248457606493886 5883.0 0.6136927304569922 154.0 0.6552489228329155

In the case of Ne1 (first column), for the observed data we have a NA, whereas for the first simulation, we have 2232. The second column corresponds to the prediction of the first NN. This is a value that ranges between 0 and 1, scaling the value of the parameter. With this data we can run the ABC analysis in R using the *abc* package. The code to get the output from each parameter is:

```
rm(list=ls());

if("abc" %in% rownames(installed.packages()) == FALSE)

{

stop("abc package is required");

}

library("abc");

abc_folder <- “path_to_abc_dl”; # CHANGE TO THE ABC_DL PATH

model.to.do <- "Model_B"; # CHANGE TO THE NAME OF THE MODEL

number.of.parameters.of.model <- 12; # CHANGE TO THE NUMBER OF PARAMETERS OF YOUR MODEL

abc.oscar <- function(target, param, sumstat, tol, method){

abc.result <- matrix(nrow=tol*nrow(sumstat),ncol=ncol(param));

colnames(abc.result) <- colnames(param);

for(col in 1:ncol(param)) {

target.s <- target[col];

sumstat.s <- sumstat[,col];

run.abc <- abc(target.s,param[,col],sumstat.s,tol = tol,method=method);

if(method=="rejection") {

if(col==1)

{

abc.result <- matrix(nrow=nrow(run.abc$unadj.values),ncol=ncol(param));

}
```

```
abc.result[,col] <- run.abc$unadj.values;
}
else {
if(col==1)
{
abc.result <- matrix(nrow=nrow(run.abc$adj.values),ncol=ncol(param));
}
abc.result[,col] <- run.abc$adj.values;
}
}
return(abc.result);
}
data.t <- read.table(file=paste(abc_folder,model.to.do,"_replication_parameter_for_abc.txt",sep=""),header=T);
data.tt <- data.t[-1,];
number.of.replicates.by.parameter <- ncol(data.t)/number.of.parameters.of.model - 1;
param <- 9; # Parameter that we want to estimate.
print(names(data.tt)[((number.of.replicates.by.parameter+1)*(param-1)+1)]);
sim.param <- data.tt[,((number.of.replicates.by.parameter+1)*(param-1)+1)];
start <- ((number.of.replicates.by.parameter+1)*(param-1)+2);
end <- ((number.of.replicates.by.parameter+1)*(param-1)+2+(number.of.replicates.by.parameter-1));
if(start!=end) {
ss.pred <- rowMeans(data.tt[,start:end]);
observed <- mean(data.t[1,start:end]);
} else {
ss.pred <- data.tt[,start];
observed <- data.t[1,start];
}
result.matrix <- abc.oscar(observed,as.matrix(sim.param),as.matrix(ss.pred),1000/nrow(data.t),"loclinear");
```

In this case, we have computed the posterior probability distribution of parameter 9, which corresponds to tSplitPop3_Pop4. *result.matrix* is a vector with 1,000 values sampled from the posterior distribution of this parameter. We can estimate centrality and dispersion statistics to describe its posterior distribution:

```
> mean(result.matrix)
```

[1] 3025.628

> quantile(result.matrix, probs = c(0.025,0.975));

2.5% 97.5%

2706.755 3333.263

Bibliography

Csilléry, K., Blum, M.G.B., Gaggiotti, O.E., and François, O. (2010). Approximate Bayesian Computation (ABC) in practice. Trends Ecol. Evol. *25*, 410–418.

Excoffier, L. and Foll, M. (2011). fastsimcoal: A continuous-time coalescent simulator of genomic diversity under arbitrarily complex evolutionary scenarios. Bioinformatics *9*, 1332—1334.Heaton, J. (2015). Encog: Library of Interchangeable Machine Learning Models for Java and C#. J. Mach. Learn. Res. *16*, 1243–1247.

Korneliussen, T.S., Albrechtsen, A., and Nielsen, R. (2014). ANGSD: Analysis of Next Generation Sequencing Data. BMC Bioinformatics *15*, 356.

Mondal, M., Bertranpetit, J., and Lao, O. (2019). Approximate Bayesian computation with deep learning supports a third archaic introgression in Asia and Oceania. Nat. Commun. *10*, 246.

Pouyet F., Aeschbacher S., Thiéry A., and Excoffier, L. (2018). Background selection and biased gene conversion affect more than 95% of the human genome and bias demographic inferences. eLife *7*.
Wong, W., Jiang, B., Wu, T., and Zheng, C. (2018). Learning Summary Statistic for Approximate Bayesian Computation via Deep Neural Network. Stat. Sin. *27*, 1595–1618.