ABC-DL pipeline for modelling and estimating demographic parameters in JAVA language and R

Introduction

The paper "*Approximate Bayesian computation with a Deep Learning algorithm (ABC-DL) supports at least a third archaic introgression common to all Asian and Oceanian human populations*" introduces a methodology that combines the statistical approach Approximate Bayesian Computation (ABC) with Deep Learning (DL). The idea is that we can use DL to train a neural network to predict a demographic model or a parameter from a demographic model, and then use the ABC framework to estimate the posterior probability of the parameter/model of the observed data. In order to show that the proposed methodology works, we implemented a pipeline in JAVA/R and applied it to the identification of archaic introgression in Asian and Oceanian human populations.

In the proposed framework, implementing a particular model comparison/parameter estimation requires JAVA and R programming skills and it is obvious that there could be other better implementations using other languages/packages. This is to say that by no way this document should be considered as "how to do ABC-DL", but just to use the implementation that we have done to a very particular problem – comparing complex demographic models that include archaic introgression.

In order to use the JAVA package, you will need the SDK1.8 and JRI1.8, as well as a JAVA editor to import the src code. The 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 the simulations. We assume that the reader has the basic knowledge to import 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 running independent jobs in the CNAG cluster. In practice, we had to 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:

1. main_folder
    a. fastSimcoal0
    b. fastSimcoal1
    c. …
    d. fastSimcoal*n*
    e. model
    f. parameter
        i. model_a
        ii. model_b
        iii. …
        iv. model_*k*

In *main_folder* we have all the information that is required to retrieve the genetic variation of the observed data in Plink binary format (please refer to https://www.cog-genomics.org/plink2/formats) and identifying the callable regions. The observed data must contain at least two individuals from each population (unless we want to use the same individual for the training and the replication) and one additional individual with name "Ancestral" that defines the ancestral allele.

We also have the fastSimcoal folders, each containing the executable of fastSimcoal2, a folder called "*model*" where the network for identifying the models are generated and the "*parameter*" folder, which contains the networks of the parameters for each model.

The observed data and the genes and CpG island files must be created before we attempt to run any analysis.

Creating a new project

The JAVA pipeline is implemented to create output files that contain the DL predictions for models or parameters from a given model in simulated and observed data. This output will be used in a second step in an ABC framework. We have used the package *abc* (https://cran.r-project.org/web/packages/abc/index.html), but there are many other packages implementing ABC that could be used.

Creating a new project will require generating an object from the class *ProjectInformation* that will contain all the information related to the exact location of the main folder, as well as the observed individuals that we are going to use for the training and replication and the populations that we want to use to generate the SFS. 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 (for example, the case of the Denisovan) the same individual is used for training and replication. This should not be a big issue, since the training only considers the observed individuals for noise injection. Furthermore, in principle, one would like to use ALL the populations for both model and parameter estimation. Nevertheless, the number of SFS cells (which are used as input in the DL) is $3^P - 2$ 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_fake_data", new ModelsToRunABC_DL(""), "C:\\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 C:\\ ABC_DL\\test_model\\, that we will use the executable *fs26* which is stored in 4 folders fastSimcoal$_x$, in a windows system, 10,000 replicates simulated at each fastSimcoal folder for each model, using as training for noise injection the training samples and the replication samples for the parameter/model estimation, using all the populations to retrieve. A key point is that we have to specify which are the models that we are going to run (*ModelsToRunABC_DL*), a list that extends the *Load_Model_Data* class (see next).

Generating a model in the ABC-DL JAVA framework

Before we can start thinking in running the ABC-DL, first we have to implement in JAVA the models that we want 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.

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. The main difference is that, instead of setting a value for an event (say, “the time of split between population 1 and population 2 is 20 generations”) it allows a range of values uniformly distributed (“the time of split between population 1 and population 2 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.

Next, we explain how to generate the model of Figure1.

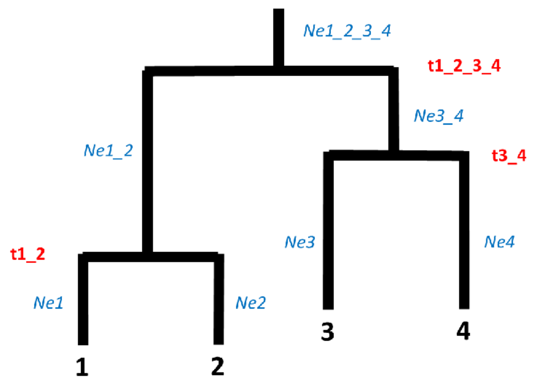

Figure 1. Model A. Ne_x = effective population size of population x. t_{x_y} = time of split between population x and y

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 case sensitive. Every time we need to call one of the populations, we will do 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. If the samples are from present times, we only have to call:

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

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

```
    demography.addSampleWithTime("Pop3", 1800, 2); //Archaic population 1 is sampled 1800 generations ago
    demography.addSampleWithTime("Pop4", 1414, 2); //Archaic population 2 is sampled 1414 generations ago
```

```
  }
```

Next, we need to specify the demographic model in the *initializeModelParameters()* method using a fastSimcoal2-like language. 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;
```

For the first population, we define an effective population size that ranges between 1000 and 5000 chromosomes.

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

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

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

And we update the demography of population 1 (Pop1) with the effective population size object Ne1

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

We do the same for the other populations:

```
// 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);
```

Now we need to add the events. An event corresponds to an interaction between populations starting (or at) a given time. In our case, every time there is an event, the effective population size of the involved populations change. 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;

// 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;
```

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)));
```

Since this is a parameter we want to estimate, 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 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 corresponds to the new growth rate, which in our case is 0 (we are considering a constant population size) and the migration matrix, which is not going to change over time because we are not considering migration events in this model. 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);
```

This piece of code is similar to the one for population 3 and 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 3 and 4. Here we use another type of distribution:

```
// 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);

}
```

Now we can implement another model, this including an introgression event between population 3 and population 2:

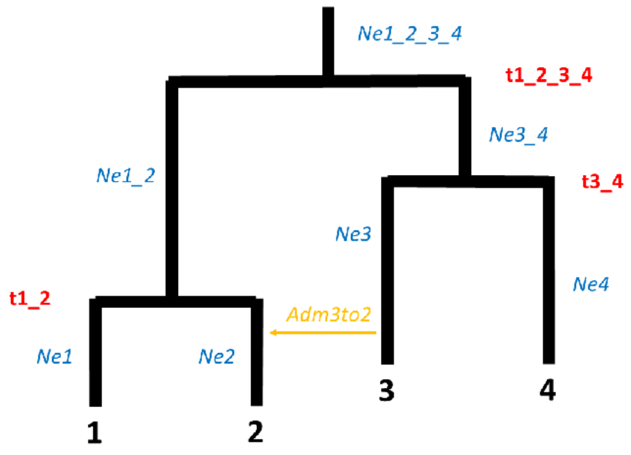

Figure 2. Model B. This model is the same as model A, but includes an introgression event between population 3 and 2.

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

// 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 that event, we defined two new parameters (the time of the introgression and the amount of introgression). The event parameter considers that population 2 sends to population 3 the amount of introgression at the time of introgression. 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. The way to specify that in the project is creating a class that extends Load_Data_Model and implements the defineModels() method:

```
public class ModelsToRunABC_DL extends Load_Model_Data{
  @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, we will generate as real data from another model:

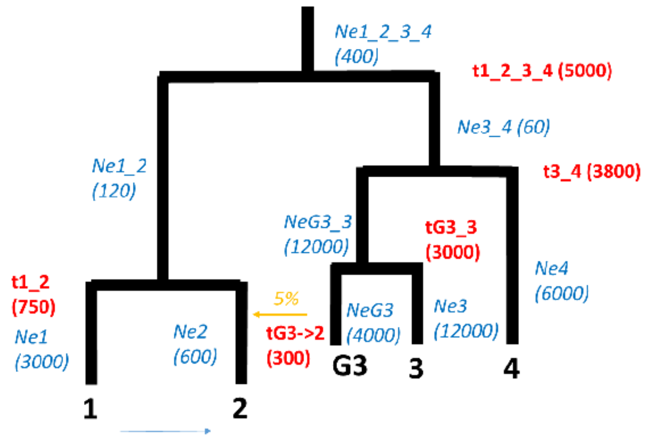

Figure 3. ModelR. The model that generated the real data. We will assume that this model is unknown, but related to a certain extent to the models that we are going to compare (Model A and Model B).

The code for this model is similar to model B, but it includes an additional Ghost population (G3) from which we do not have samples, and an admixture event between Ne1_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);

    demography.addSampleWithTime("Pop3", 1800, 4); //Archaic population 1 is sampled 1800 generations ago

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

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

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

      demography.addSample(names[pop], 4);

    }

    demography.addSample("G3",0);

  }


  @Override

  protected void initializeModelParameters() throws ParameterException {

    // 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 must be specified in the 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);
```

Running simulations

Simulations are 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 the simulation, we will need to specify which are the fragments we want to simulate, 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 fragments, each of 10Mb (notice that a REAL project will use the whole genome). Within each fragment, we are going to consider all the region as 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

This means that fragment one, which comprises 10Mb, has two callable regions. 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 fastSimcoalx 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 4 folders fastSimcoal 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 fastSimcoal0 folder.

If everything works out, in the fastSimcoal0 folder, after executing the main method, one should see that a file with name output_Model_A.txt has been created in the folder fastSimcoal0 and data from simulations is being stored.

The first row contains the names of the *k* parameters of Model_A that we are considering, using the names that we defined when specifying each Parameter. The next row is the result of a simulation. The first *k* columns correspond to the values that we have used for the simulation. The next $3^P - 2$ for P populations corresponds to the multidimensional unfolded site frequency spectrum, as 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. We do this calling:

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

Where model is the number of the model (in our example, 0 or 1), and 0.5 defines the number of simulations that are going to be used for training of the Neural Network and which are going to be used for the replication (ABC analysis). This will generate two text output files for each model in the main folder.

Generating the neural networks for model prediction

Once we have generated the simulated data for the different models, we need to generate the neural network(s) that predict the model given the SFS. When doing this, we need to include as noise injection the SFS from real data 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. Since the observed data contains two samples from each population plus a sample called Ancestral that indicates which is the ancestral allele, we can use one sample from each population to be used as noise injection during the training of the networks.

Generating a neural network to predict to which model the SFS belongs is obtained by running

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

The first number corresponds to the error threshold for stopping the training of the neural network. The second number corresponds to the id of the network. The third parameter is the project information and the last one, the number of neurons at each intermediate layer.

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 that is not achieved after 6 hours or 1000 iterations, the network training will stop.

If everything was fine, an Encog neural network file will be created in the folder model, with the format:

```
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 reporting a neural network. Now, we want to make the model prediction of the observed data not used for the noise injection during the training, as well as of these simulations that we have not used for training the NNs, stored in the "output_x_replication.txt" files.

In order to generate the prediction output, we call

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

This will create a file in the working directory called model_predictions.txt. Each row will contain the prediction for the K models that we have considered in the N neural networks that are stored in the model folder. The total number of rows will correspond 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.

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

```
observed 3.118074844506109E-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
```

For the first NN, the observed data has a probability of ~0 of being Model_A and ~1 of being Model_B, for the last, a similar prediction. For the first simulation of Model_A, the probability of being A is 0.88 for the first NN and 0.78 for the last trained NN.

For the first simulation of Model_B, we see that the first NN assigns it to Model_B with a probability of ~1; the same for the las NN.

Using this output as observed summary statistics, we can conduct an Approximate Bayesian Computation approach to compute the posterior probability of each model given the observed data. In our implementation, we used abc package. The script to run the abc model selection is:

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

library("abc");

# now, for each of the neural network replicates, compute the Bhattachardyya 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.t <- read.table(file="C:\\ABC_DL\\test_model\\model_predictions.txt",header=F);
observed.ss <- data.t[which(data.t[,1]=="observed"),2:ncol(data.t)]
simulated.ss <- as.matrix(data.t[-which(data.t[,1]=="observed"),2:ncol(data.t)]);
model.names <- levels(as.factor(as.character(data.t[-which(data.t[,1]=="observed"),1])));
models <- rep(-1,nrow(simulated.ss));
models.by.row <- data.t[-which(data.t[,1]=="observed"),1];
for(m in 1:length(model.names))
{
  models[models.by.row==model.names[m]] <- m;
}
mean.ss <- compute.mean.nn(simulated.ss, 10, 2);
mean.observed.ss <- compute.mean.nn(observed.ss,10,2)
res <- postpr(mean.observed.ss, models, mean.ss,tol=1000/nrow(simulated.ss),method="mnlogistic");
```

If we run this code, we will 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 of the Parameters of Model B

Now that we have identified model B as the one showing the largest posterior probability, the next question would be to compute the posterior probability of each of the parameters of that model. Of each parameter, we can run different replicates. In order to generate the replicate of the neural network of a parameter from a model, we call:

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

We need to pass the project information, which model we want to consider (following the example, we want to use Model_B, which in the list of models is the second, which 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 and the error threshold to stop).

After running, a file storing the trained Encog network will be generated:

…\Parameter\Model_B\ name_parameter_replicate.network

Once we have generated several replicates of a NN for each parameter, we can make the prediction of each parameter in the replication dataset. To do this, we call:

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

The first parameter is the project information. The second, the model we want to use. And the third is the number of replicates we have done of each parameter of the model that we are considering. If we try to use more replicates for each parameter and/or we are using a model where we have not trained the neural networks of the parameters, we will get nasty IO errors.

After running this, we will 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 names of the parameters. The next row to the prediction with the observed data. For each parameter, the first column has value NA, because we do not know which is the value of the parameter that generated the data. The next rows corresponds to the parameter values of the simulated data and the predicted values. For example, in our case, an output could look like this:

```
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

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
```

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

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:

```
library("abc");

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);

}


 model.to.do <- "Model_B";
 number.of.parameters.of.model <- 12;

 data.t <- read.table(file=paste("C:\\Users\\olao\\OneDrive - CRG - Centre de Regulacio Genomica\\ABC_DL\\test_model\\",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;

 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 this 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
```