

sBayes documentation

Peter Ranacher, Nico Neureiter, Natalia Chousou-Polydouri, Olga Sozinova

May 6, 2021

Contents

Introduction	3
How to install sBayes	3
Linux (Debian/Ubuntu)	3
MacOS	3
Windows	4
Data coding	5
Preparing the data	5
The <i>features.csv</i> file	5
Model parameters	7
Confounding effects	7
Number of areas	7
Priors	7
Universal preference, inheritance, and contact	8
Weights	9
The geo-prior	9
Size of an area	10
MCMC and diagnostics	11
Warm-up	11
Sampling from the posterior	11
Diagnostics	12
Visualizing the posterior	13
Maps	13
Density plots (for universal preference, inheritance and contact)	13
Weight plots	14
DIC plots	14
A step-by-step analysis in sBayes	16
The <i>config.JSON</i> file	16
data	17
model	18
prior	18
mcmc	23

results	25
Inference	26
Post-processing	26
Plotting	26
Simulation	26
The log file	26
Get Data	28
Read empirical data	28
Features	28
Prior counts	29
Cost matrix	30
Update the log file	31
Simulate data	32
Run the simulation	32
Update the log file	35
MCMC	35
Set-up the MCMC	35
Model parameters	35
Sampling parameters	36
Sampling	38
Evaluate the ground truth	38
Update the log file	38
Save Results	38
Plot maps and statistics	39
Initialize plotting	39
Load input files	39
Plot types	40
Map	40
Weights	40
Probabilities	40
DIC	40
Multiple plots	40

Introduction

This document explains how to use **sBayes** - a software package for finding areas of language contact in space, based on linguistic features. Contact areas comprise geographically proximate languages, which are similar and whose similarity cannot be explained by the confounding effect of universal preference and inheritance. For more details, see [LINK TO PUBLICATION](#). A typical analysis in **sBayes** consists of five steps:

1. Data coding and preparing the features file
2. Defining the model and the priors
3. Setting up and running the MCMC
4. Summarizing and visualizing the posterior sample

Each of the steps above will be covered in detail in the following sections of the manual. **sBayes** can also be used for simulation of linguistic areas and linguistic families. Simulations will be covered in the last section of the manual.

How to install sBayes

In order to install **sBayes**, you need to have Python3 on your computer. You can check what version of python you have by opening a terminal and typing `python`. Then you can install **sBayes**. The exact steps to do this depend on your operating system (due to different ways of installing dependencies). Following are the instructions for Linux, MacOS and Windows.

Linux (Debian/Ubuntu)

To install **sBayes**, open a terminal window, navigate to the folder where you want to install it and follow these steps:

1. Get the **sBayes** source code by running `git clone https://github.com/derpetermann/sBayes`. This will create a **sBayes** directory on your computer.
2. Navigate to this new directory by running `cd sBayes`.
3. Install GEOS, GDAL and PROJ by running `sudo apt-get install -y libproj-dev proj-data proj-bin libproj-dev`.
4. Install **sBayes** along with some required python libraries by running `pip install .`

MacOS

On MacOS we recommend using homebrew to install the required system packages. Open a terminal window, navigate to the folder where you want to install **sBayes** and follow these steps:

1. Get the **sBayes** source code by running `git clone https://github.com/derpetermann/sBayes`. This will create a **sBayes** directory on your computer.
2. Navigate to this new directory by running `cd sBayes`.
3. Install GEOS, GDAL and PROJ by running `brew install proj geos gdal`
4. Install **sBayes** along with some required python libraries by running `pip install .`

Windows

On Windows we recommend using <https://www.anaconda.com/Anaconda> to install the required packages. To do this, download Anaconda from <https://www.anaconda.com/>, open an Anaconda terminal window, navigate to the folder where you want to install **sBayes** and follow these steps:

1. Get the **sBayes** source code by running `git clone https://github.com/derpetermann/sBayes`. This will create a **sBayes** directory on your computer.
2. Navigate to this new directory by running `cd sBayes`.
3. Install GEOS, GDAL and PROJ manually or using the OSGeo4W installer <https://trac.osgeo.org/osgeo4w>.
4. Install **sBayes** along with some required python libraries by running `pip install .`

Data coding

sBayes receives as input a matrix of discrete categorical data in the format of a CSV file. The precise structure of the *features.csv* file will be covered below.

Preparing the data

The data for **sBayes** consist of a number of features, each with a number of *states*. Each feature is a linguistic property presumed relevant for the discovery of areas and should be logically independent from other features (i.e. there shouldn't be a common causal mechanism linking the two features). For example, treating a /p/-/b/ contrast, a /t/-/d/ contrast and a /k/-/g/ contrast as three independent features is tripling what arguably is a single voiceless-voiced stop contrast. Using logically dependent features or strongly correlated features results in an inflation of confidence, as it is essentially counting multiple times the same underlying feature).

Each feature has a number of mutually exclusive discrete states. Features can be binary or multistate. It should be noted that every state is considered as potentially contributing to a linguistic area, and therefore all states should be "informative" or forming a "natural" class. For example, a feature about vowel inventory size with two states: "three vowels" and "not three vowels" is problematic, because the state "not three vowels" is a non-informative state. There are multiple ways of not having an inventory of three vowels and there is no good argument that three-vowel inventories spread and are inherited, while all other vowel inventories are similar enough and could have come about through inheritance or contact irrespective of the number of their vowels. An alternative in this case would be to have a state for every observed number of vowels in an inventory. In general, states based on the negation of another state and absent states are usual suspects for being non-informative and should be examined in detail.

The *features.csv* file

The *features.csv* has the following structure:

Table 1: Columns in *features.csv*

Column		Description
<i>id</i>	required	a unique identifier for each language
<i>x</i>	required	the spatial x-coordinate of the language
<i>y</i>	required	the spatial y-coordinate of the language
<i>family</i>		the name of a family or sub-clade of a family
< <i>F1</i> >		feature 1
< <i>F2</i> >		feature 2
...		

$\langle F1 \rangle$, $\langle F2 \rangle$, ... are placeholders for feature names, e.g. *vowel_inventory_size*. Feature names must be unique. Distinct entries in each feature column are treated as distinct categorical states of the feature, i.e. “1”, “2”, “two” and “2.0” in *vowel_inventory_size* are interpreted as four distinct categories or four different inventory sizes. To avoid mix-ups users can provide all applicable states of any given feature in the *feature_states.csv* file (see below). A blank space in the CSV – leaving a cell empty – is interpreted as NA.

Geographically, languages are represented by a point location, e.g. their centre of gravity. Coordinates x and y uniquely define the location of a language, either in a spherical coordinate reference system (CRS), i.e. latitude longitude, or projected to a plane, e.g. a Universal Transverse Mercator CRS. If spherical CRS is used, x corresponds to the longitude and y to the latitude. We are not surveyors, after all.

sBayes models inheritance per distinct entry in the column *family*. Languages with the same entry for *family* are assumed to belong to a common family (or sub-clade of a family). If left empty, inheritance is not modelled for this language. Here is an example of a *features.csv* file together with the applicable states for all features in *feature_states.csv*.

Table 2: Example of *features.csv*

id	name	x	y	family	f1	f2	f3
des	Desano	-1092883.205	3570461.932	Tucanoan	N	A	Y
tuo	Tucano	-1102073.299	3569861.124	Tucanoan	N	B	
trn	Trinitario	-557535.8929	1836033.262	Arawak	Y	C	N
...							

Table 3: All applicable states in the corresponding *feature_states.csv*

f1	f2	f3
N	A	N
Y	B	Y
	C	
	D	

Model parameters

sBayes aims to predict why language l has feature f with state s , e.g. why retroflex affricates (f) are present (s) in the Arawakan language Chamicuro (l). **sBayes** proposes three explanations – the feature is universally preferred, has been inherited in the family or adopted through contact. Similarities due to universal preference and inheritance are treated as confounding effects, while the remaining similarities are attributed to contact: **sBayes** proposes myriads of different areas, such that their similarities cannot be explained by confounding. **sBayes** infers one categorical distribution for universal preference, inheritance in each family, an contact in each area per feature f .

Confounding effects

sBayes infers the assignment of languages to contact areas from similarities in the data that are not due to confounding effects. Universal preference is always modelled as a confounder, while the effect of inheritance can be turned on or off by the analyst (the default is on). If inheritance is modelled as a confounder, languages must be assigned to families (or sub-families) in *features.csv*.

Number of areas

The number of contact areas n defines how many distinct areal groupings are assumed in the data. We suggest to run the algorithm with $n = 1$, increase n iteratively, and evaluate the posterior evidence of each model in post-processing, for example using the deviance information criterion (DIC).

Priors

The following parameters in **sBayes** require a prior distribution:

- universal preference, inheritance (preference in each family) and contact (preference in each area) per feature
- weights per feature
- spatial allocation of an area, i.e. the geo-prior
- size of an area

The prior for universal preference and inheritance and the geo-prior can be informed empirically. In what follows we introduce the prior for each parameter and we explain how prior knowledge can be informed empirically.

Universal preference, inheritance, and contact

The prior for universal preference (α_f) expresses our knowledge about the global preference of a state before seeing the data. Similar, the prior for inheritance ($\beta_{f,\phi}$) expresses our knowledge about the preference of a state in a language family, for example Arawak. While these priors can be uniform, i.e. each state is equally probably a-priori, this might not appropriately reflect our knowledge about language. Some states might be more common, e.g. because they are essential for communication or easily processed in our brains or inherited in a family. Languages will be more likely to share these states, which requires a re-tuning of the confounding effect. We can empirically inform the prior for universal preference and inheritance, for example from a global language database. **sBayes** models the prior for universal preference with a Dirichlet distribution (the prior for inheritance in a family can be defined analogously):

$$P(\alpha_f) = \text{Dir}(\psi_i = 1 + \mu_i \cdot \rho) \quad \text{for } i \in 1, \dots, k,$$

μ_i is the prior probability of state i and defines the mean of the prior distribution. ρ gives the precision or inverse variance. ψ_i can be thought of as pseudocounts for state i . A large precision yields large pseudocounts and a strong prior. When $\rho = 0$, $\psi_i = 1$ for all i and the prior is uniform. The mean μ_i can be derived empirically, either from

- aggregated counts (i.e. a counts file)
- relative frequencies (i.e. a relative frequency file),
- features (a feature file similar to *features.csv*)

The precision ρ_i is estimated from the aggregated counts or feature file, or set by the analyst. Note that the prior is always defined before seeing the data. In other words, aggregated counts, relative frequencies or features must not contain information about any of the languages in the sample. Table 4 shows the structure of the counts / relative frequency file. Table 19 shows universal counts for the features in *features.csv*:

Table 4: Columns in a counts/ relative frequency file

Column	Description
<i>feature</i>	required
<i>feature</i>	a unique identifier for each feature
<S1>	empirical counts/relative frequency for state 1
<S2>	empirical counts/relative frequency for state 2
...	

The *feature* column contains all distinct feature names, the remaining columns store the observed counts (or relative frequencies) for each state. <S1>, <S2> are placeholder for the names of the states. If a state is not applicable for a feature the corresponding cell is left empty.

Table 5: Example of *universal_counts.csv*

feature	A	B	C	N	Y
f1				30	4
f2	10	9	15		
f3				16	18
...					

Before seeing the data, we usually do not know where contact areas are or which features languages share in these areas. Thus, the prior for contact, i.e the preference in each area is uniform.

Weights

The prior on weights (w_f) is uniform, i.e. a priori universal preference, inheritance and contact are equally likely:

$$P(w_f) = \text{Dir}(\psi_1 = 1, \psi_2 = 1, \psi_3 = 1) .$$

The geo-prior

The geo-prior models the *a priori* probability of languages to be in contact, given their spatial locations. A *uniform* geo-prior assumes all areas to be equally likely, irrespective of their location in space. The *cost-based* geo-prior builds on the assumption that geographically proximate languages are more likely to be in contact than distant ones. Distance is modelled as a cost function C , which assigns a non-negative value $c_{i,j}$ to each pair of locations i and j . By default costs are expressed as Euclidean distance (for planar coordinates) or the great-circle distance (for spherical coordinates).

Alternatively, users can provide a cost matrix to quantify the effort to traverse geographical space (e.g. in terms of hiking effort, travel times, ...). Since costs are used to delineate contact areas, they are assumed to be symmetric, hence $c_{i,j} = c_{j,i}$. For cost matrices where this is not immediately satisfied (such as hiking effort), the costs are made symmetric, e.g. by averaging the entries for $c_{i,j}$ and $c_{j,i}$ in the cost matrix. The cost matrix has the following structure:

Table 6: Columns in *cost_matrix.csv*

Column	Description
<i>language</i>	a unique identifier for each language
<i><language 1></i>	distances to language 1
<i><language 2></i>	distances to language 2
...	

<language 1>, *<language 2>* are placeholder for the unique identifiers of language 1 and 2. The element (i, j) in the CSV gives the costs to travel from entity i to entity j . The cost matrix has a trace of zero, i.e. the distance of each entity to itself is zero. The matrix is not necessarily symmetric. The costs (i, j) might differ from the costs (j, i) . Table 7 shows an example of a cost matrix, again corresponding to data in *features.csv*.

Table 7: Example of *cost_matrix.csv*

id	des	tuo	trn
des	0	100	40
tuo	100	0	15
trn	40	15	0

Size of an area

In addition to the geo-prior, there is an implicit prior probability on $|Z|$, the number of languages in an area $Z \in \mathcal{Z}$. **sBayes** employs two types of priors for $|Z|$. The *uniform area prior* assumes that each area is equally likely a-priori. This puts an implicit prior on size, such that larger $|Z|$ are preferred over smaller ones: there are $(M - m + 1)/m$ more ways to choose m objects from a population M than $m - 1$ objects, given $m \leq M/2$. The *uniform size prior* assumes that all $|Z|$ are equally likely a-priori, i.e. $P(|Z|)$ has a uniform prior in the interval $[\min(|Z|), \max(|Z|)]$.

MCMC and diagnostics

sBayes adopts a Markov Chain Monte Carlo (MCMC) approach to sample from the posterior distribution of a model. In the warm-up phase, multiple independent chains explore the parameter space in parallel. After warm-up, **sBayes** moves to the chain with the highest likelihood from where it starts to sample from the posterior. The analyst can tune both the warm-up and the actual sampling.

Warm-up

For the warm-up, the analyst can define the number of iterations and the number of chains to explore the parameter space. Each sampler starts from a random initial location. More steps and more chains increase the chances that the warm-up reaches a high density region from where the actual sampling can take over, but at the same time result in a longer run time. What is an appropriate number of steps and chains depends on the complexity of the model, i.e. the number of languages, features, families and areas. If the analyst does not set the number of iterations and chains, **sBayes** falls back on default values. These might not be appropriate for complex models. We also recommend to run the algorithm several times (e.g. five) and compare the posterior distribution across runs. Together with the warm-up, this lowers the possibility that an unfortunate starting location has an influence on the posterior sample.

Sampling from the posterior

For the actual sampling, the analyst can set the number of iterations (i.e. the steps in the Markov chain), the sample size and the operators used in the MCMC. The number of iterations depends on the complexity of the model. CAUTION: If not defined, **sBayes** uses default values, which might not be appropriate for complex models. The sample size defines the number of posterior sample that are returned to the user. In an MCMC, consecutive steps in the chain are necessarily auto-correlated. Only sufficient iterations and an appropriate ratio between samples and steps guarantees that consecutive samples in the posterior are uncorrelated. We discuss below how to estimate the effective sample size (ESS), i.e. the number uncorrelated, independent samples in the posterior.

The user can define how the MCMC proposes new samples, i.e. which operators are used and how often each operator is called. **sBayes** employs spatial operators to propose new areas (i.e. to add, remove or swap languages in an area) and a Dirichlet proposal distributions to propose new weights and estimates for universal preference, inheritance and contact per feature. For the Dirichlet proposal distribution, the user can define the precision of each proposal step, i.e. how far away from the current sample are new candidates recruited. While a high precision ensures that the parameter space is explored exhaustively, it also makes sampling more ineffective: **sBayes** needs more iterations to collect sufficient independent samples.

An analysis in **sBayes** returns the following output:

- `areas*.txt`: posterior samples of contact areas
- `stats*.txt`: posterior samples of all remaining parameters, prior, posterior and log-likelihood of each sample
- `log.txt`: a log file with statistics about the run.

Diagnostics

sBayes offers different diagnostic tools to monitor progress and assess the quality of the posterior sample:

- Progress: During sampling, on-screen messages inform about the current progress and return the likelihood of the model after each 1000 steps.
- Acceptance/rejection statistics: The log file provides acceptance\rejection statistics for each operator. A high acceptance rate indicates too frequent and, thus, inefficient sampling, a low acceptance rate indicates too sparse sampling.
- Convergence and effective sample size: The results in stats*.txt are compatible with **Tracer** (<https://beast.community/tracer>), a software package for visualising and analysing the MCMC trace files generated through Bayesian phylogenetic inference. Tracer provides trace plots to assess convergence and ESS statistics to assess correlation in the posterior. Tracer shows if an MCMC has converged to a stable distribution and collected sufficient independent samples, but it does not tell whether the sampler got stuck in a local optimum. We recommend to run **sBayes** several times (eg. five), use Tracer to check for convergence, and compare both the posterior probability and the posterior estimates across runs.

Visualizing the posterior

`sBayes` offers built-in plotting functions to visualize the posterior samples. There are four main types of plots: maps, density plots, weight plots, and DIC plots.

Maps

Maps show the posterior distribution of contact areas in geographic space, i.e. the spatial location of all languages (dots), their assignment to contact areas (colored dots and lines) and, optionally, to families (colored polygons). Dot size indicates how often a language is in an area. Languages, which appear together in the same area and which neighbors in a Gabriel graph, are connected with a line. Line thickness indicates how often two languages are together in the posterior. Users can add different legend items and include an overview map (see example in figure 1).

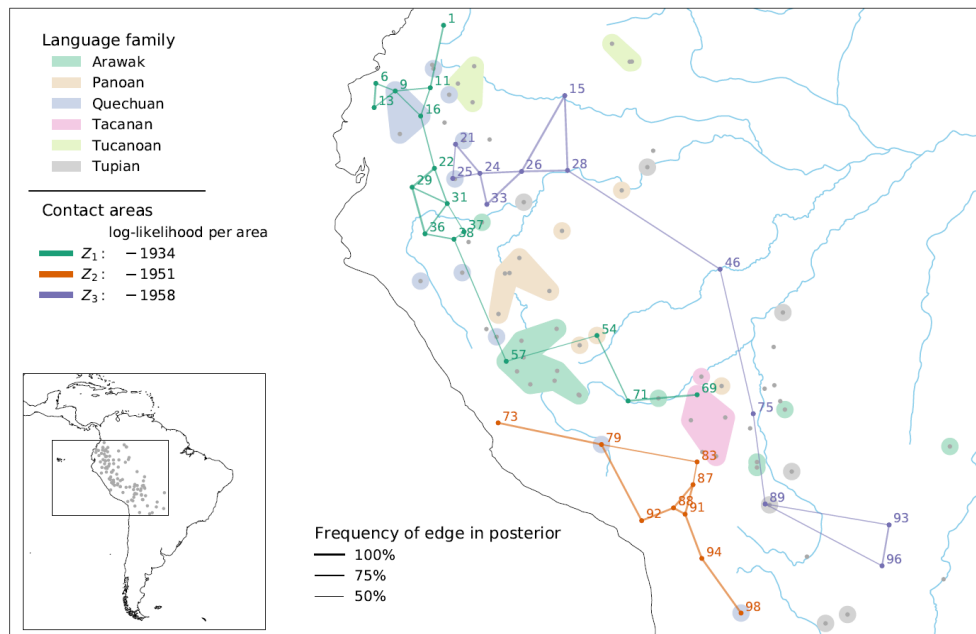


Figure 1: A map with three contact areas (green, orange and purple dots and lines).

Density plots (for universal preference, inheritance and contact)

These plots visualize the preference for each of the states of a feature, either universally, in a family or a contact area. The appearance of the plot changes depending on the number of states: densities are displayed as ridge plots for two states (see Figure 2), in a triangular probability simplex for three states (similar to the weights, see next section), a square for four states, a pentagon for five, and so on. `sBayes` returns the density plots for all features per family or area or globally, in a single grid. Figure 2 shows the density plot for features F1, F2

with two states (N, Y) in an area. While the posterior distribution for F1 in the area is only weakly informative, with a slight tendency for Y, F2 clearly tends towards state N.

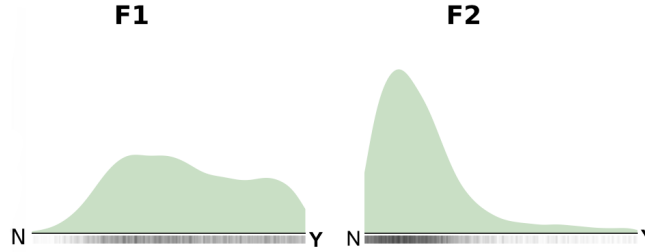


Figure 2: The density plot shows the posterior preference for two features (F1, F2) in an area

Weight plots

Weight plots visualize the posterior densities of the weights per feature: how well does each effect – universal preference, inheritance and contact – predict the distribution of the feature in the data? The densities are displayed in a triangular probability simplex, where the left lower corner is the weight for contact (C), the right lower corner the weight for inheritance (I), and the upper corner the weight for universal preference (U). Figure 3 shows the weight plot for two features - F24 and F26. The distribution of F24 is best explained by inheritance and contact – both receive high posterior weights, but there is no single best explanation for F16 – the posterior weights are all over the place. The pink dot marks the mean of the distribution (optional). Again, **sBayes** returns the density plots for all features in a single grid.

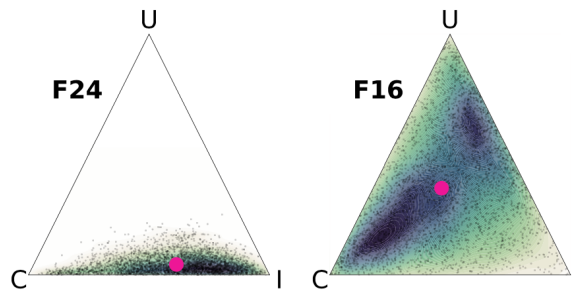


Figure 3: Weight plots for two features (F24, F16).

DIC plots

The Deviance Information criterion (DIC) is a measure for the performance of a model, considering both model fit and model complexity. DIC plots visualize the DIC across several models, usually with increasing number of areas (n), and help the analyst to decide for an appropriate

number of areas. As a rule of thumb, the best model is the one where the DIC levels off. Figure 4 shows the DIC for seven models with increasing number of areas – $n = 1$ to $n = 7$. The DIC levels off for $n = 4$, suggesting four salient contact areas in the data. As the DIC plot compares performance across models, it needs several result files as input.

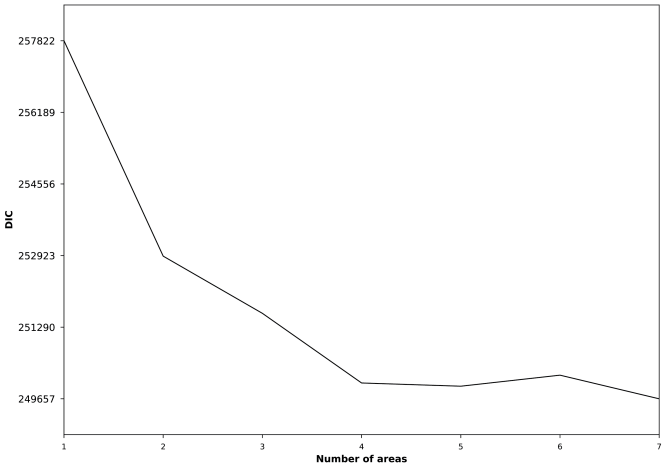


Figure 4: DIC plot for models with increasing number of areas ($n = 1$ to $n = 7$)

A step-by-step analysis in sBayes

A default analysis in **sBayes** consists of three main steps:

- ***Configuration***

The user collects the data in the *features.csv* file and defines the settings for the analysis in the *config.JSON* file.

- ***Inference***

The user passes both the data and the settings to **sBayes** and runs the MCMC

- ***Post-processing***

The user can explore and visualize the posterior distribution, either directly in **sBayes**, with third-party applications – such as **Tracer**, or own plotting functions.

The *config.JSON* file

Users define the settings of a **sBayes**-analysis in the *config.JSON* file. The *config.JSON* file has four main parameters (or keys):

- **data**: provides file paths to the empirical data
- **model**: defines the likelihood, the prior, and additional model parameters
- **mcmc**: gives the settings for the Markov chain Monte Carlo sampling
- **results**: gives the location of the name of the result files

data

In **data**, the user provides the file paths to the *features.csv* file (**features**) and the applicable states for all features (**feature_states**). Users can give absolute or relative file paths. Relative paths are assumed to start from the location of the *config.JSON* file. The following JSON snippet tells **sBayes** to open the file *balkan_features.csv*, with applicable states in *balkan_features_states.csv*. Both files are located in the sub-folder **data** (relative to *config.JSON*).

```
"data": {  
  "features" : "data/balkan_features.csv"  
  "feature_states": "data/balkan_features_states.csv"}  
}
```

Table 8 summarizes all parameters in *data* and gives the default values and expected data types. (*required*) indicates that a parameter is mandatory and has to be set by the user.

Table 8: The *config.JSON* file: parameters in **data**

parameter	data type	default value	description
features	string	(required)	file path to the <i>features.csv</i> file
feature_states	string	-	file path to the <i>feature_states.csv</i> file

model

In `model`, users define the likelihood function and additional model parameters. Users can specify whether or not the model considers inheritance (`inheritance`), give the number of contact areas (`n_areas`), and define the minimum and maximum number of languages in an area (`min_m`, `max_m`). The parameter `prior` is discussed in detail below. Note that the minimum

The following JSON snippet defines a model which considers inheritance, has five areas and a minimum and maximum size of 10 and 40 languages per area.

```
"model": {
  "inheritance": true,
  "n_areas": 5,
  "min_m": 10,
  "max_m": 40,
  "prior": {
    ...
  }
}
```

Table 9 summarizes all parameters in `model` and gives the default values and expected data types.

Table 9: The *config.JSON* file: parameters in `model`

parameter	data type	default value	
<code>inheritance</code>	boolean	(required)	Does the model consider inheritance?
<code>n_areas</code>	number	(required)	Number of contact areas in the model
<code>min_m</code>	number	3	Minimum number of languages per area
<code>max_m</code>	number	50	Maximum number of languages per area
<code>prior</code>	JSON		Defines the prior of the model. See below.

prior

In `prior`, the user provides the prior distribution for each parameter in the model. There are six different types of priors (see, section Priors) and hence six parameters in `prior`:

- `universal`: the prior for universal preference
- `inheritance`: the prior for inheritance or preference in a family
- `contact`: the prior for contact or preference in an area
- `weights`: the prior for the weights
- `geo`: the geo-prior

- **area_size**: the prior on area size

Each parameter takes as input a JSON object, which – depending on the type of prior – can itself have several sub-parameters (for a full list see, Table 10).

universal

The prior for universal preference (**universal**) takes as input a JSON object with the following parameters: **type**, **file**, **file_type**, and **scale_counts**. The default **type** is *uniform*, which defines a uniform Dirichlet prior distribution, such that for each feature each state is equally likely a priori. In case of a uniform prior **file** no other parameters need to be provided.

When users have additional information to inform the Dirichlet distribution, they can set **type** to *informed_dirichlet* and provide the path to a **file** with the information to construct the prior. **sBayes** supports different formats to encode the information in a file (see section Priors). When the user provides Dirichlet priors

When the information is empirical and comes from aggregated counts or relative frequencies from outside the sample **file_type** is set to *counts_file*. When the information is in the same format as the *features.csv* **file_type** is set to *features_file*.

The mean of the Dirichlet distribution is directly inferred from the empirical information in **file**. The variance is either inferred directly too, or it can be scaled using **scale_counts**. This is useful when aggregated counts give a plausible expectation for the states, but result in an overly strong prior. Small values for (**scale_counts**) define a weakly informative prior, while large values give a strong prior with low variance. Since frequencies can only give the mean of the Dirichlet distribution, **scale_counts** must be provided when relative frequencies are provided in the *counts_file*.

The following JSON snippet defines an empirically informed Dirichlet prior for universal preference. The empirical information to construct the prior is read from the file *universal_preference.csv*, which has the same file structure as the *features.csv* file. The prior is scaled to ten counts and, thus, weakly informative.

```
"prior": {
  ...
  "universal": {
    "type": "empirical",
    "file": "universal_preference.csv",
    "file_type": "features_file",
    "scale_counts": 10}
}
```

inheritance

The prior for preference in a family (**inheritance**) takes as input a JSON object with the following parameters: **type**, **files**, **file_type**, and **scale_counts**. The default **type** is *uniform*, which defines a uniform Dirichlet prior distribution, such that in each family and for

each feature each state is equally likely a priori. In this case the user does not need to provide any additional parameters. In case of an *empirical* prior for inheritance, the user must provide information analogous to the prior for universal preference. However, there is one important difference: languages in the sample might come from several families. Thus, **files** provides the file paths to several files, one per family. To enable unambiguous matching, each key in **files** takes one of the family names in the *family* column in *features.csv* and points to one file with empirical information per family.

The following JSON snippet defines an empirically informed prior for preference in two families, Arawak and Tucanoan. The information to construct the prior is provided in a *counts_file* per family, *arawak_counts.csv* and *tucanoan_counts.csv*. The parameter **scale_counts** is not given and the prior is not scaled.

```
"prior": {
  ...
  "inheritance": {
    "type": "empirical",
    "files": {
      "Arawak": "arawak_counts.csv",
      "Tucanoan": "tucanoan_counts.csv"},
    "file_type": "counts_file"}
}
```

contact and weights

The priors for preference in an area (**contact**) and the **weights** are currently always set to *uniform*.

geo

The **geo** prior takes as input a JSON object with the following parameters: **type**, **file**, **covariance**, and **rate_exp**. The default prior is of type *uniform*, in which case every spatial allocation of points in an area has the same prior probability. In this case, the user does not need to provide any additional parameters. Other possible types are *Gaussian* and *cost_based*.

For the Gaussian prior, the spatial locations in an area are evaluated against a two-dimensional Gaussian distribution, for which the user provides **covariance**, a 2×2 variance-covariance matrix with units km^2 . The code snippet below defines a Gaussian geo-prior with a variance in *x* and *y* of 100 km^2 and a covariance of zero, i.e. the bivariate normal distribution is perfectly spherical.

```
"prior": {
  ...
```

```

"geo": {
  "type": "Gaussian",
  "covariance": [[100, 0],
                 [0, 100]]}
}

```

For the cost-based geo prior, `sBayes` computes the minimum spanning tree (MST) between all locations in each area. The MST expresses the least costs necessary for all languages in the area to have been in contact. By default costs are expressed as either Euclidean distance or distance on a sphere. Which of the two is used depends on the coordinate reference system of the input locations. Alternatively, users can provide the file path to a cost matrix (`file`) quantifying the effort to travel between all pairs of languages. In this case, the MST will find the path with the least costs connecting all locations in the area. The MST is then evaluated against an exponential decay function with rate `rate_exp`. The rate gives the mean expected costs between events of language contact. It defines the rate at which the probability for contact decreases with increasing costs. The code snippet below enforces a cost-based geo prior. The costs are given in the cost matrix *travel_times.csv* as travel times in hours. The rate is set to 12 hours.

```

"prior": {
  ...
  "geo": {
    "type": "cost_based",
    "file": "travel_times.csv",
    "rate_exp": 12}
}

```

area_size

There are two types of priors for area size: *uniform_area* which defines a prior that is uniform over all areas (introducing a bias towards larger areas). *uniform_size* enforces a prior distribution that is uniform over all sizes of an area (for details see section). The following JSON snippet defines a prior that is uniform over size.

```

"prior": {
  ...
  "area_size": {
    "type": "uniform_size"}
}

```

Table 10 summarizes all prior parameters, gives the default values and expected data types.

Table 10: The *config.JSON* file: parameters in **prior**

parameter	data type	default value	description
universal	JSON	-	The prior for universal preference
type	string	"uniform"	The prior is either <i>uniform</i> or <i>empirical</i>
file	string	-	File path to the empirical data
file_type	string	-	File format of the empirical data
scale_counts	number	-	Scale factor for the empirical data
inheritance	JSON	-	The prior for preference in a family
type	string	"uniform"	The prior is either <i>uniform</i> or <i>empirical</i>
files	JSON	-	File paths to the empirical data
family 1	string	-	File path to the empirical data for family 1
family 2	string	-	File path to the empirical data for family 2
...		-	
file_type	string	-	File format of the empirical data
scale_counts	number	-	Scale factor for the empirical data
contact	JSON	-	The prior for preference in an area
type	string	"uniform"	Currently only <i>uniform</i> priors are supported
weights	JSON	-	The weights prior
type	string	"uniform"	Currently only <i>uniform</i> priors are supported
geo	JSON	-	The geo-prior
type	string	"uniform"	The geo-prior is either <i>uniform</i> , <i>Gaussian</i> , or <i>cost_based</i>
covariance	array	-	The covariance matrix for the Gaussian distribution
file	string	-	The file path to the cost matrix.
rate_exp	number	-	Rate of the exponential distribution to evaluate the distance-based geo-prior.
area_size	JSON	-	The prior on area size.
type	string	"uniform_area"	The prior on area size. Either <i>uniform_area</i> or <i>uniform_size</i> .

mcmc

In *mcmc* users define how **sBayes** samples from the posterior distribution. The parameter **n_runs** gives the number of independent MCMC runs for the same model. Each run generates an independent posterior sample. In postprocessing, users can then check if the posterior samples across all runs converge to the same stable distribution.

Each run starts with a **warmup**. During warmup, several independent chains search the parameter space in parallel to find regions of relative high density, which helps the main MCMC to sample efficiently. Users can define the number of warmup chains **n_warmup_chains** and the number of steps in each chain (**n_warmup_steps**). Both depend on the complexity of the model: complex models with many languages, many parameters and many states per parameter need more warmup chains and steps than simple models with few languages and few parameters. All samples generated during warmup are discarded.

After warmup, the main MCMC takes over and samples from the posterior distribution. Users can define the number of steps in the Markov chain (**n_steps**) and the number of samples retained from the chain **n_samples**, i.e. the size of the posterior sample. Again, the number of steps depends on the complexity of the model.

The following code snippet defines an MCMC analysis with 5 independent runs. Each run takes 1,000,000 steps and collects 10,000 posterior samples. In the warmup phase, 20 independent chains take 100,000 steps to search for high density regions in the area.

```
"mcmc": {  
  ...  
  "n_runs": 5,  
  "n_steps": 1000000,  
  "n_samples": 10000,  
  "warmup": {  
    "n_warmup_chains": 20,  
    "n_warmup_steps": 100000  
  }  
}
```

During sampling, the MCMC algorithm picks operators to change different parameter values of the model. Some operators change the areas, others the weights and yet others the universal preference or the preference in a family or an area. **steps** defines how often each operator is called. Values are provided as relative frequencies and need to add up to 1. Moreover, users can give the initial size of an area (**initial_area_size**) and define how often the proposal distribution for growing areas is spatially informed (**p_grow_connected**), i.e. how often do areas grow to adjacent languages in the auxiliary graph and how often do they grow to random, not necessarily adjacent languages. Note that **initial_area_size** must be set such that the Markov chain is irreducible, which is only the case when the **initial_area_size** times the number of areas is smaller than the number of languages in the sample.

In the following code operators to modify areas (grow, shrink, swap) and universal preference are called in 10% of all steps, operators to modify inheritance in 20% and those to change the

weights and contact in 30%. The initial size for areas is 6. 85% of all steps grow to adjacent languages.

```
"mcmc": {
  ...
  "steps": {
    "area": 0.1,
    "weights": 0.3,
    "universal": 0.1,
    "inheritance": 0.2,
    "contact": 0.3},
  "p_grow_connected": 0.85,
  "initial_area_size": 6
}
```

Table 11 summarizes all mcmc settings, gives the default values and expected data types. Note that the default values for `n_steps` and `n_warmup_steps` might be much too low for complex models and also other MCMC parameter depend on model characteristics. Users should always verify that the MCMC has converged to a stable distribution and that it has created sufficient independent samples for each parameter.

Table 11: The *config.JSON* file: parameters in mcmc

parameter	data type	default value	description
<code>n_runs</code>	number	1	Number of independent runs of the analysis
<code>n_steps</code>	number	100000	Number of steps in the Markov chain
<code>n_samples</code>	number	1000	Number of samples in the posterior
<code>warmup</code>	JSON	-	Settings for the warmup
<code>n_warmup_chains</code>	number	15	Number of warmup chains
<code>n_warmup_steps</code>	number	100000	Number of warmup steps
<code>steps</code>	JSON	-	Operators and their frequencies
<code>area</code>	number	0.05	Frequency of area steps
<code>weights</code>	number	0.4	Frequency of weights steps
<code>universal</code>	number	0.05	Frequency of universal steps
<code>inheritance</code>	number	0.1	Frequency of inheritance steps
<code>contact</code>	number	0.4	Frequency of contact steps
<code>p_grow_connected</code>	number	0.85	Frequency of grow steps to adjacent languages
<code>m_initial</code>	number	5	Number of languages in initial areas

results

In **results** users provide the name and the file location of the results file. The parameter **results_path** gives the relative or absolute file path to the folder where the results will be saved. **log_file** creates a log file with meta information about the analysis, i.e. model parameters and acceptance//rejection statistics for each operator. The parameter **file_info** defines which additional information about the analysis is passed to the file name. The default is "n", which adds the number of areas to the file name. Alternatively, "i" adds information about whether or not inheritance was considered in the model.

The following code snippet creates a folder *results* relative to the location of *config.JSON* file. **sBayes** returns a log file and information about the number of areas is added to the result files.

```
"results": {
  "results_path": "results",
  "log_file": true,
  "file_info": "n"
}
```

Table 12 summarizes all parameters in **results**, gives the default values and expected data types.

Table 12: The *config.JSON* file: parameters in **results**

parameter	data type	default value	
results_path	string	"results"	Relative or absolute file location to save the result
log_file	boolean	true	Return a log file?
file_inf	string	"m"	either "m" or "i" meta information a added to the file name

Inference

Post-processing

Plotting

Simulation

Table 13 summarizes all parameters of the config file and lists the default values and expected data types. *(required)* indicates that a parameter is mandatory and has to be set by the user. All other parameters are optional and if omitted are set to the default values. Parameters will be discussed in detail where they appear in the model.

The log file

Last, we create a log file to save the relevant metadata associated with the experiment.

```
exp.log_experiment()
```

When the analysis has finished running, the log file will be returned together with the results and tell us what the experiment was all about. Our future selves will thank us later.

Table 13: Parameters in the config file

	parameter	data type	default value	description
mcmc	N_AREAS	int	(required)	Number of areas
	INHERITANCE	bool	(required)	Is inheritance modelled as a confounder?
	PRIOR	dict	(required)	Prior probability for all parameters
	N_STEPS	int	30,000	Number of steps in the MCMC
	N_SAMPLES	int	1,000	Number of samples in the posterior
	BURN_IN	int	5,000	Number of steps discarded as burn-in
	N_RUNS	int	1	Number of times the MCMC is run
	MIN_M	int	3	Minimum number of languages in an area
	MAX_M	int	200	Maximum number of languages in an area
	PROPOSAL_PRECISION	dict	see below	Defines the width of the proposal distribution
	STEPS	dict	see below	Defines how often each sampling operator is called
	N_CHAINS	int	5	Number of parallel chains in the MCMC
	SWAP_PERIOD	int	1,000	Steps between two attempted chain swaps
	N_SWAPS	int	3	Number of attempted chain swaps
	NEIGHBOR_DIST	string	"euclidean"	Distance function in the neighborhood graph
results	LAMBDA_GEO_PRIOR	real	see below	Decrease of contact probability with increasing costs
	RESULTS_PATH	string	"../results"	File path to results folder
data	FILE_INFO	str	"n"	Additional information shown in the file name
	FEATURES	str	(required)	File path to features
	PRIOR_COUNTS	dict	see below	File paths to count data for priors
simulation	COST_MATRIX	str	null	File path to cost matrix
	INHERITANCE	bool	(required)	Is inheritance simulated?
	SITES	str	(required)	File path to sites for which contact is simulated
	AREA	int	(required)	Area for which contact is simulated
	E_CONTACT	real	(required)	Simulated entropy of contact
	I_CONTACT	real	(required)	Simulated intensity of contact
	E_UNIVERSAL	real	1.0	Simulated entropy of universal pressure
	I_UNIVERSAL	real	1.0	Simulated intensity of universal pressure
	E_INHERITANCE*	real	2	Simulated entropy of inheritance
	I_INHERITANCE*	real	0.2	Simulated intensity of inheritance
	N_FEATURES	int	35	Number of simulated features
	P_N_STATES	dict	see below	Number of simulated states
	SUBSET	bool	false	Return only a subset of the simulated data?

PROPOSAL_PRECISION: {"weights": 30, "universal": 30, "contact": "30", "inheritance"*: 30}

PRIOR_COUNTS: {"universal": null, "inheritance"*: null}

P_N_STATES: {"2": 0.4, "3": 0.3, "4": 0.3}

STEPS: {"area": 0.05, "weights": 0.65, "universal": 0.05, "contact": 0.2, "inheritance"†: 0.05}

LAMBDA_GEO_PRIOR‡: "auto_tune"

*omitted if inheritance is not considered in the model

†omitted if inheritance is not considered in the model, other values are re-normalized to sum to 1

‡omitted if the geo-prior is "uniform" (see section MCMC)

Get Data

Read empirical data

The class **Data** reads feature and prior data from CSV files and converts them to a format that **sBayes** can interpret. First we call the class:

```
from src.load_data import Data
dat = Data(experiment=exp)
```

The argument *experiment* passes the metadata associated with the current experiment to the class, specifically the "data" parameters in the config file, which tell **sBayes** where to look for data.

Features

Next, we read the features from the CSV file:

```
dat.load_features()
```

load_features opens a CSV file whose path is given in "data" in the config file. Suppose the features are in the file *features.csv*, which is in the folder *../data/features*. We can load the features with the following entry in the config file:

```
"data": {
  "FEATURES" : "../data/features/features.csv"
  ...
}
```

features.csv has the following structure:

$\langle F1 \rangle$ is a placeholder for the name of feature 1. Names must uniquely identify a feature. Features are assumed to be attributes of an entity with discrete, mutually exclusive states. All distinct entries in each feature column are treated as different states, for example, "1", "2", "two" and "2.0" are four different states. A blank space in the CSV – leaving a cell empty – is interpreted as NA. If spherical coordinates are provided, x corresponds to the longitude and y to the latitude. We are not surveyors, after all.

Languages with the same entry for *family* are assumed to belong to the same family (or sub-clade of a family). **sBayes** models inheritance per distinct entry in *family*. If left empty, inheritance is not modelled for that language. Here is an example of a *features.csv* file:

Table 14: Columns in *features.csv*

Column		Description
<i>id</i>	required	a unique identifier for each entity
<i>x</i>	required	the spatial x-coordinate of an entity
<i>y</i>	required	the spatial y-coordinate of an entity
<i>family</i>	required	the name of a family or sub-clade of a family
< <i>F1</i> >		feature 1
< <i>F2</i> >		feature 2
...		

Table 15: Example of *features.csv*

id	name	x	y	family	f1	f2	f3
des	Desano	-1092883.205	3570461.932	Tucanoan	N	A	Y
tuo	Tucano	-1102073.299	3569861.124	Tucanoan	N	B	
trn	Trinitario	-557535.8929	1836033.262	Arawak	Y	C	N
...							

Prior counts

`sBayes` allows for empirical priors on universal pressure and inheritance. Priors are derived from count data per state and feature, as observed in a language family or universally. Loading the count data and converting them to fully-operational Dirichlet prior distributions is straightforward:

```
dat.load_universal_counts()
dat.load_inheritance_counts()
```

`load_universal_counts` opens a CSV with universal count data whose file path is given in the config file. `load_inheritance_counts` opens a CSV with count data for each family in *features.csv*. Again, the file paths are given in the config file in a dictionary with one key per family. Suppose we want to load counts for universal pressure and inheritance for the two families in table 15. This requires the following entry in the config file:

```
"data":{
  "PRIOR_COUNTS" : {
```

```

    "universal": "../data/prior/universal_counts.csv",
    "inheritance": {
        "Tucanoan": "../data/prior/tucanoan_counts.csv",
        "Arawak": "../data/prior/arawak_counts.csv"
    }
}
...
}

```

If a family is missing in "inheritance" a flat prior is used. All *_counts.csv files have the following structure:

Table 16: Columns in *_counts.csv

Column	Description
<i>feature</i>	required a unique identifier for each feature
<S1>	counts for state 1
<S2>	counts for state 2
...	

The CSV contains a matrix with row names corresponding to the features in *features.csv* and column names corresponding to all applicable states. The feature column contains all feature names, the remaining columns the observed counts per state. <S1> is a placeholder for the name of state 1. If a state is not applicable for a feature the corresponding cell is left empty, which is interpreted as NA. The example in table 19 shows universal counts for the features in table 15:

Table 17: Example of *universal_counts.csv*

feature	A	B	C	N	Y
f1				30	4
f2	10	9	15		
f3				16	18
...					

Cost matrix

The COST_MATRIX gives the costs between all entities in *features.csv*. Costs can be expressed in terms of distance, hiking effort, travel times or any other meaningful property quantifying

the effort to traverse geographical space. If `COST_MATRIX` is left empty, costs are computed from the coordinates in *features.csv* using `NEIGHBOR_DIST` – the distance function in the neighborhood graph, which is the Euclidean distance for planar coordinates or the Great-circle distance for spherical coordinates. The script below reads the *cost_matrix.csv*:

```
dat.load_cost_matrix()
```

This is the corresponding entry in the config file:

```
"data":{
  "COST_MATRIX" : {../data/cost_matrix.csv}
  ...
}
```

The cost matrix has the following columns: where $\langle id1 \rangle$ is a placeholder for the id of entity 1:

Table 18: Columns in *cost_matrix.csv*

Column	Description
<i>id</i>	a unique identifier for each entity
$\langle id1 \rangle$	distances to entity 1
$\langle id2 \rangle$	distances to entity 2
...	

$\langle id1 \rangle$ is a placeholder for the id of entity 1. Treating *id* as row names, the element (i, j) in the CSV gives the costs to travel from entity *i* to entity *j*. Note that the cost matrix must have a trace of zero, i.e. the distance of each entity to itself is zero. The matrix is not necessarily symmetric, i.e. the costs (i, j) might differ from the costs (j, i) . Here is an example of a *cost_matrix.csv* file, again corresponding to the features in table 15:

Update the log file

Finally, we update the log file and add metadata about the features, prior counts and the cost matrix:

```
dat.log_loading()
```

Table 19: Example of *cost_matrix.csv*

id	des	tuo	trn
des	0	100	40
tuo	100	0	15
trn	40	15	0

Simulate data

In addition to empirical data, **sBayes** can be applied to simulated data, for example to compare inferred contact areas against ground truth. To simulate contact areas we first call the class *Simulation*:

```
from src.simulation import Simulation
sim = Simulation(experiment=exp)
```

The argument *experiment* passes all the metadata associated with the current experiment to *Simulation*, together with the information in the config file, which tells **sBayes** how to simulate contact areas. Then we perform the simulation:

```
sim.run_simulation()
```

run_simulation uses the parameters in the config file to simulate contact areas. The following section will go through all steps in the simulation. Parameters will be explained as they occur.

Run the simulation

First, **sBayes** reads the file *sites.csv* whose file path is given in SITES in the config file. *sites.csv* provides dummy locations in (geographic) space, which serve as a canvas to simulate contact areas.

```
"simulation": {
    "SITES" : "../simulation/sites.csv"
    ...
}
```

sites.csv has the following columns:

Sites with the same value for *area* belong to a common contact area, those with the same value for *family* to a common family. When *area* or *family* are left empty a site does not belong to a contact area or a family, respectively. All sites for which *subset* is true (i.e. 1) belong to the subset, a part of the data located in a distinct spatial region. The subset allows to test how the algorithm performs when only limited (and potentially biased) data is available for analysis. For all simulation experiments *sites.csv* was created in QGIS. Sites were manually assigned to areas, families and the subset. Table 21 shows an example of a *sites.csv* file:

Table 20: Columns in *sites.csv*

Column		Description
<i>id</i>	required	a unique identifier for each site
<i>x</i>	required	the spatial x-coordinate of a site
<i>y</i>	required	the spatial y-coordinate of a site
<i>area</i>	required	To which area does the site belong?
<i>family</i>		To which family does the site belong?
<i>subset</i>		Does the site belong to the subset?

Table 21: Example of *sites.csv*

id	x	y	area	family	subset
0	9614.58747	5215.48901	1	1	false
1	9534.5381	5216.58741	1		false
2	9190.03603	5668.6972	1	2	false
3	9537.05393	4592.57165		2	false
4	10571.79423	1182.44303	2	2	true
...					

sBayes is now ready to simulate data using the parameters provided in "simulation" in the config file. AREA gives the id of the area in *sites.csv* for which contact is simulated. AREA can be an integer or a tuple of integers, in which case contact is simulated for several areas. If INHERITANCE is true, inheritance is simulated for all families in *sites.csv*. N_FEATURES gives the number of simulated features and P_N_STATES the number of simulated states per feature:

```
"simulation": {
  "AREA" : (2, 3)
  "INHERITANCE": true,
  "N_FEATURES": 35,
  "P_N_CATEGORIES": {
    "2": 0.7,
    "3": 0.2,
    "4": 0.1
  }
}
```

```
}
```

The example above simulates contact for areas 2 and 3 in *sites.csv* as well as inheritance in all of the families. In total 35 features will be simulated of which 70% have two states, 20% three states and 10% four states.

There are two parameters which control how features are simulated in contact areas, families and universally: intensity and entropy. Intensity defines the number of features affected by either contact, inheritance or universal pressure. Entropy regulates the predictability of each effect. For example, large intensity values for contact result in many features in a contact area for which the influence of contact is strong. Small (!) entropy values for contact result in little variability: many members of the contact area are similar. Taken together, the effect of contact is strong when both intensity is large and entropy is small. The same applies to inheritance and universal pressure. Here are the corresponding parameters for intensity and entropy in the config file:

- I_CONTACT controls the intensity of contact
- E_CONTACT controls the entropy of contact
- I_UNIVERSAL controls the intensity of universal pressure
- E_UNIVERSAL controls the entropy of universal pressure
- I_INHERITANCE controls the intensity of inheritance
- E_INHERITANCE controls the entropy of inheritance

E_INHERITANCE and I_INHERITANCE are only applicable if inheritance is simulated. The values for intensity and entropy are not directly interpretable. They are passed as pseudocounts to a set of Dirichlet distributions and define how peaky and concentrated these distributions are. This in turn gives the intensity and entropy of contact, inheritance and universal pressure in the simulated data. While suitable values for intensity and entropy can be found in the example below, we recommend to inspect the simulated data and iteratively tune the parameters.

```
"simulation": {  
  "I_CONTACT": 2,  
  "E_CONTACT": 0.25,  
  "I_UNIVERSAL": 1.  
  "E_UNIVERSAL": 1.  
  "I_INHERITANCE": 3.  
  "E_INHERITANCE": 0.1  
  ...  
}
```

In the above example, inheritance is simulated to have the strongest effect (low entropy, high intensity), followed by contact. Universal pressure is simulated to have a weak effect (high entropy, low intensity).

Using the above parameters the algorithm samples from the likelihood to generate simulated data. When SUBSET is true only the simulated data in the subset are returned. The remaining data are used to derive simulated priors (see section MCMC).

Update the log file

We can update the log file and add metadata about the simulation:

```
sim.log()
```

MCMC

This is the core of **sBayes**. First, we define the Bayesian clustering model, set-up the MCMC and pass it the data and parameters from the config file. Then we sample from the posterior distribution. For simulated data, we evaluate the ground truth, i.e. the posterior of the true contact area.

Set-up the MCMC

To initialize the MCMC we call the class *MCMC* and pass both the data and the experiment, which tells **sBayes** where to find the parameters.

```
from src.mcmc_setup import MCMC
mc = MCMC(data=dat, experiment=exp)
```

When analysing simulated data – e.g. *sim* in the above example – we simply pass these to *data*. There are two types of mcmc parameters in the config file:

- model parameters
- and sampling parameters

Model parameters

The model parameters tune the likelihood function and the priors. Specifically, N_AREAS gives the number of distinct contact areas in the model. INHERITANCE defines whether inheritance is modelled as a confounder.

PRIOR gives the prior distribution for all parameters in a dictionary, with keys for the geo-prior ("geo"), the weights ("weights"), universal pressure ("universal"), inheritance ("inheritance") and contact ("contact"). The default is "uniform", which models a uniform Dirichlet distribution for the weights, universal pressure, inheritance and contact and a discrete uniform distribution over all areas for the geo-prior. Alternatively, the prior for universal pressure and inheritance can be set to "from_counts", which uses the data in PRIOR_COUNTS as pseudocounts for defining a Dirichlet distribution. For simulated data (and if SUBSET is true), we can also set the prior for universal pressure to "from_simulated_counts", in which case the complement of the subset will be used to simulate priors. The geo-prior can be set to "cost_based", in which case the costs along the minimum spanning tree are evaluated against an exponential distribution: areas with high costs have low prior probability. Cost are either derived from the cost matrix (see section Cost matrix) or - if omitted - from the distance function in NEIGHBOR_DIST (see next paragraph). The parameter LAMBDA_GEO_PRIOR

defines the exponential distribution, which by default is automatically tuned from either the cost matrix or the Euclidean distance matrix ("auto_tune"). For weights and contact only uniform priors are applicable.

In addition to the prior distributions, there are two implicit parameters relating to the prior probability of contact areas: MIN_M and MAX_M define the minimum and maximum number of languages per contact area.

```
"mcmc": {
  "N_AREAS": 2,
  "INHERITANCE": true,
  "MIN_M": "3",
  "MAX_M": "100",
  "PRIOR": {
    "geo": "cost_based",
    "weights": "uniform",
    "universal": "from_counts",
    "inheritance": "from_counts",
    "contact": "uniform"
  },
  "LAMBDA_GEO_PRIOR": "auto_tune",
  ...
}
```

The above parameters create a model with two contact areas, which can each comprise a minimum of three and a maximum of 100 languages. The model considers inheritance. The prior for weights and contact are uniform, the prior for universal pressure and inheritance are derived from count data. The geo-prior is cost-based: the costs of an area are evaluated against an auto-tuned exponential distribution.

Sampling parameters

The sampling parameters control how `sBayes` samples from the posterior distribution. `N_STEPS` gives the number of steps in the Markov chain. The default is 30,000, which might be way too few steps for models with many parameters. We strongly recommend to monitor the convergence of the MCMC (see section Plot results and statistics) and adjust `N_STEPS` accordingly. `BURN_IN` gives the number of initial steps discarded as burn-in. Again, the default of 5,000 might not be appropriate for larger models. `N_SAMPLES` defines the number of samples in the posterior distribution. `N_RUNS` gives the number of times the analysis is repeated. Results are only reliable when several independent runs converge to the same posterior. In the example below, the MCMC will run three times with 100,000 steps each, of which 2,000 are sampled and the first 10,000 are discarded as burn-in.

```
"mcmc": {
  "N_STEPS": 100000,
  "N_RUNS": 3,
  "N_SAMPLES": 2000,
  "BURN_IN": 10000,
  ...
}
```

```
}
```

STEPS give the proportion of how often each operator is being called during sampling. The parameter contains a dictionary with keys for each operator: "area" for growing, shrinking or swapping an area, "weights" for changing the weights, "universal", "inheritance" and "contact" for changing the parameter vectors of each of the three effects. The default values for STEPS are shown below:

```
"mcmc": {
  "STEPS": {
    "area": 0.05,
    "weights": 0.7,
    "universal": 0.05,
    "contact": 0.2
  }
  ...
}
```

The key "inheritance" is only applicable if the model considers inheritance. If not the default parameters are adjusted such that they to sum to 1. PROPOSAL_PRECISION controls how new candidates are sampled for the weights and the parameter vectors (universal pressure, inheritance and contact). The parameter provides pseudocounts for tuning a Dirichlet distribution: large values result in proposals which are close to the current sample, small values allow for longer steps. PROPOSAL_PRECISION contains a dictionary with the following keys:

```
"mcmc": {
  "PROPOSAL_PRECISION": {
    "weights": 30,
    "universal": 30,
    "contact": 30,
    "inheritance": 30,
  }
  ...
}
```

The next three parameters relate to the Metropolis Coupled Markov Chain Monte Carlo (MC³). **sBayes** employs several independent chains to explore the parameter space in parallel and, thus, sample more efficiently. N_CHAINS gives the number of chains in the MC³. Too few chains might miss high density regions in the posterior, while too many chains slow down the analysis. SWAP_PERIOD gives the interval (in terms of steps) between two attempted chain swaps and N_SWAPS defines the number of attempted swaps. We recommend to use the default values shown below:

```
"mcmc": {
  "N_CHAINS": 5,
  "SWAP_PERIOD": 1000,
  "N_SWAPS": 3,
```

```
...  
}
```

Finally, `NEIGHBOR_DIST` defines the distance metric for the Delauney triangulation and the neighborhood graph. The default is "euclidean". If the data have geographic coordinates (i.e. latitude and longitude) "great_circle" should be used instead.

Sampling

Once the model and the MCMC set-up are ready sampling is straight-forward:

```
mc.sample()
```

We can pass an *initial_sample* to the function, which will start the MCMC at a predefined location in the parameter space. If left empty the sampling will start from a random location. Moreover, if *lh_per_area* is set to true (default), the contribution of each of the `N_AREAS` to the likelihood will be evaluated in post-processing.

Evaluate the ground truth

When analyzing simulated data, we can evaluate the ground truth, the likelihood and the posterior of the true contact area, which will help to assess the performance of the algorithm:

```
mc.eval_ground_truth()
```

Update the log file

Last, we write the metadata for both the model and the sampling statistics (e.g. the acceptance ratios) to the log file:

```
mc.log_setup()  
mc.log_statistics()
```

Save Results

When the MCMC has finished running, we save the results and the log file in the folder given in `RESULTS_PATH` in the config file:

```
mc.save_samples()
```

Results are saved as pickle files. `FILE_INFO` in the config file tells *save_samples* which metadata to attach to the file name. The default is "n" which will give the number of areas in the model. Other possible values are "i" indicating whether inheritance was considered in the model, "p" indicating whether informed priors were used and "s", which is only meaningful for simulated data and gives the strength of the simulated signal and the id of the simulated area. Moreover, the current run is automatically added to the file name. Taken together a possible pickled result file could look like this:

```
RESULTS_PATH\contact_areas_n2_1.pkl
```

This implies that there are two contact areas in the posterior and the first run of the model was pickled.

Finally, the log file is saved to

```
RESULTS_PATH\experiment.log
```

Plot maps and statistics

Initialize plotting

The class **Plot** reads the data and the results of the experiment and plots maps and statistics: probabilities, weights and the deviance information criterion (DIC).

First, we instantiate an object of the class **Plot**:

```
from sbayes.plot import Plot
plot = Plot(simulated_data=False)
```

The argument *simulated_data* is true in case of an experiment with simulated data and is false when analyzing the real world data.

Load input files

Then, we load the needed input files, namely, configuration settings, data and results. The paths to the data and the results files should be listed in the configuration file:

```
"path_data": "../data/features/features.csv",
"path_results": "../results",
"path_areas": [
    "../results/n1/areas_n1.txt",
    "../results/n2/areas_n2.txt"
],
"path_stats": [
    "../results/n1/stats_n1.txt",
    "../results/n2/stats_n2.txt"
],
```

After adjusting the config file, we can load it and read the data:

```
plot.load_config(config_file='../results/config_plot.json')
plot.read_data()
```

If we want to plot the results of only one model, e.g. n1, we need to call the function *read_results* with an argument *model* set to 'n1':

```
plot.read_results(model='n1')
```

In case we want to produce plots for all the models, we need to get the list of the models' names and call the above mentioned function in a loop:

```
model_names = plot.get_model_names()
for model_name in model_names:
    plot.read_results(model=model_name)
```

Plot types

Map

In order to generate a posterior map, we have to call this function:

```
plot.posterior_map()
```

The function has three optional arguments: *file_name*, *file_format* and *return_correspondence*. The first two arguments determine the name and the extension of the file, where the map is saved. By default it is 'mst_posterior.pdf'.

The argument *return_correspondence* specifies an additional output of the languages' labels in a separate table. By default it is set to False.

Weights

In order to analyze the distribution of weights, we can generate a weights grid for each feature using this function:

```
plot.plot_weights_grid()
```

Probabilities

The probability distribution of the binary features can be analyzed with a probability grid:

```
plot.plot_probability_grid()
```

DIC

If we have multiple models, we can calculate a deviance information criterion (DIC) for each model, in order to see which one should be preferred:

```
plot.plot_dic(results_per_model)
```

The function *plot_dic* takes as an argument a dictionary of all models, for which the DIC is calculated.

In order to collect the results of all models, we can run the following loop:

```
results_per_model = {}  
model_names = plot.get_model_names()  
for model_name in model_names:  
    results_per_model[model_name] = plot.results
```

Figure 4 shows that the model with six contact areas has the lowest DIC value, thus, it should be more plausible than the other models.

Multiple plots

In most experiments, it is needed to produce multiple plots of one type using different parameter values. For example, if we want to see how the posterior map changes with the different minimum posterior frequency, we can write the following code:


```

for model_name in model_names:
    plot.read_results(model=model_name)
    min_posterior_frequency = [0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3]
    for mpf in min_posterior_frequency:
        plot.config['map']['content']['min_posterior_frequency'] = mpf
        plot.posterior_map(file_name='posterior_map_' + model_name
                             + '_' + str(mpf))

```