sBayes documentation

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

May 20, 2021

Contents

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

<F1>, *<F2>*, ... 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 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 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):

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

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 5 shows universal counts for the features in *features.csv*:

Table 4: Columns in a counts/ relative frequency file

Column		Description
feature	required	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:

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

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
language	a unique identifier for each language
<*language 1*>	distances to language 1
<*language 2*>	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).

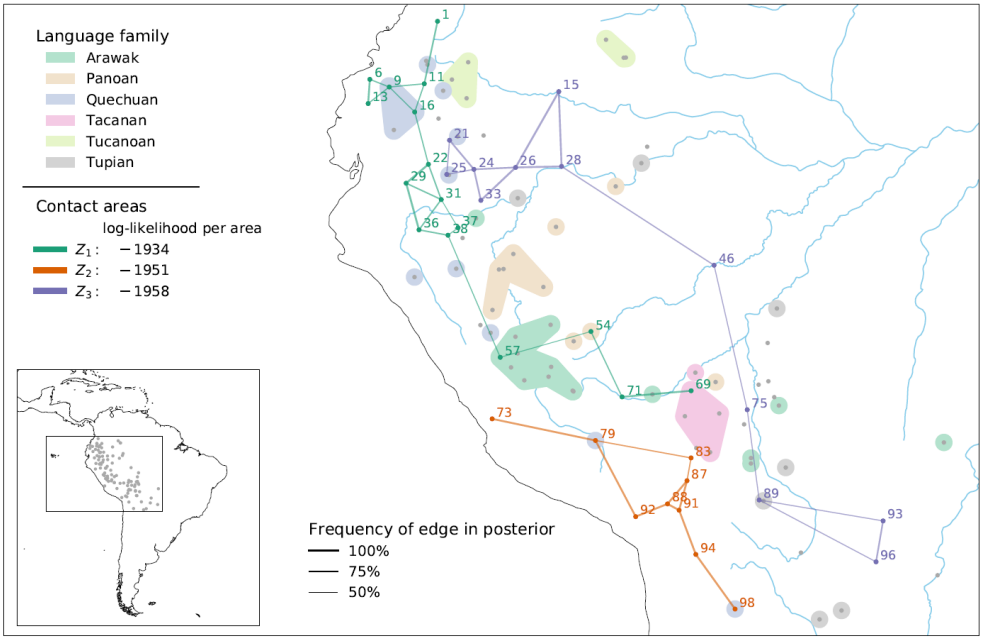

Figure 1: A map with three contact areas (green, orange and purple dots and lines).

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.

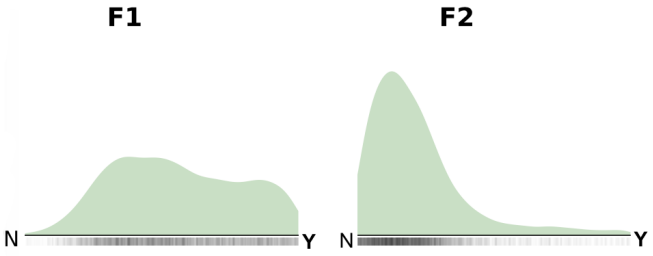

Figure 2: The density plot shows the posterior preference for two features (F1, F2) in an area

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.

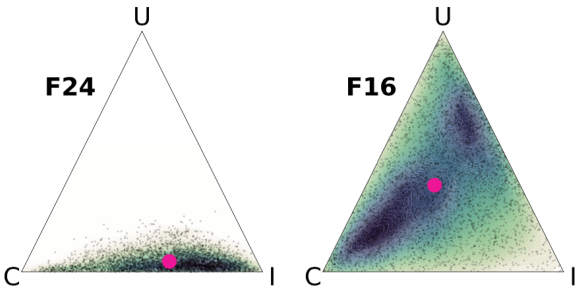

Figure 3: Weight plots for two features (F24, F16).

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.

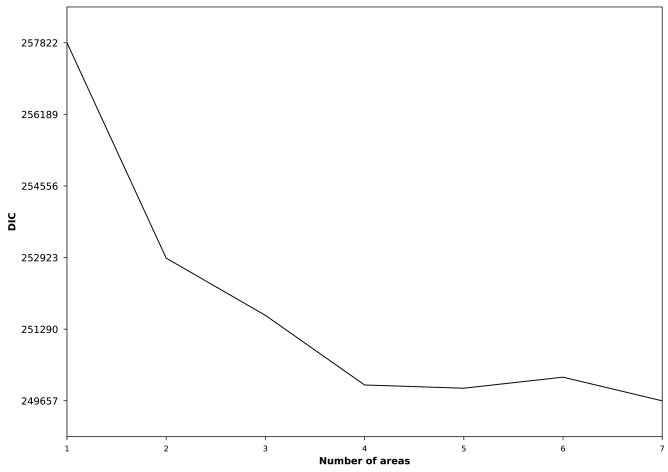

Figure 4: DIC plot for models with increasing number of areas ($n = 1$ to $n = 7$)

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 *features.csv* file
`feature_states`	string	-	file path to the *feature_states.csv* 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,
  "min_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**	
`inheritance`	boolean	(required)	Does the model consider inheritance?
`n_areas`	number	(required)	Number of contact areas in the model
`min_m`	number	3	Minimum number of languages per area
`max_m`	number	50	Maximum number of languages per ares
`prior`	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 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 *uniform* or *empirical*
`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 *uniform* or *empirical*
`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 *uniform* priors are supported
`weights`	JSON	-	The weights prior
`type`	string	"uniform"	Currently only *uniform* priors are supported
`geo`	JSON	-	The geo-prior
`type`	string	"uniform"	The geo-prior is either *uniform*, *Gaussian*, or *cost_ based*
`covariance`	array	-	The covariance matrix for the Gaussian distringibution
`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 *uniform_ area* or *uniform_ size*.

`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 `mcmcm` 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 MCMCM 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**
`n_runs`	number	1	Number of independent runs of the analysis
`n_steps`	number	100000	Number of steps in the Markov chain
`n_samples`	number	1000	Number of samples in the posterior
`warmup`	JSON	-	Settings for the warmup
`n_warmup_chains`	number	15	Number of warmup chains
`n_warmup_steps`	number	100000	Number of warmup steps
`steps`	JSON	-	Operators and their frequencies
`area`	number	0.05	Frequency of area steps
`weights`	number	0.4	Frequency of weights steps
`universal`	number	0.05	Frequency of universal steps
`inheritance`	number	0.1	Frequency of inheritance steps
`contact`	number	0.4	Frequency of contact steps
`p_grow_connected`	number	0.85	Frequency of grow steps to adjacent languages
`m_initial`	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