sBayes documentation

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

August 23, 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 the original 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 (four different inventory sizes). To avoid potential mix-ups and assignments to nonexistent states, 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 as latitude and longitude in a spherical coordinate reference system (CRS), or projected to a plane, for example, the 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.

Table 2 shows an example of a *features.csv*-file. Table 3 gives the applicable states for the three features in Table 2: person affixes for possession (*pers_aff*) are either Y (present) or N (absent), participant roles marked on verb (*role_mark*) can take states A (A marking), B (P marking), c (both A and P marking), D (either A or P marking), or E (neither A nor P marking), and phonemic oral-nasal contrast for vowels (*vow_con*) is Y (present) or N (absent).

Table 2: Example of *features.csv*

id	name	x	y	family	pers_aff	role_mark	vow_con
des	Desano	-1092883.205	3570461.932	Tucanoan	N	A	Y
tuo	Tucano	-1102073.299	3569861.124	Tucanoan	N	A	Y
cot	Caquinte	-1423618.645	2206737.923	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	
	E	

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, testing whether languages in these areas areas are similar and whether these similarities can be explained by confounding. `sBayes`infers one categorical distribution for universal preference, inheritance in each family, an contact in each area per feature f. All model parameters are passed to `sBayes` in a separate configuration file (see, section about the *config.JSON* file).

Confounding effects

`sBayes` infers the assignment of languages to contact areas from similarities in the data that are poorly explained by the 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 K defines how many distinct areal groupings are assumed in the data. We suggest to run the algorithm with $K = 1$, increase K 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 from empirical data. In what follows we introduce the prior for each parameter.

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. Both priors can be uniform – each state is equally probable a-priori – or reflect empirical knowledge about language: some states might be more common, because they are essential for communication, 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. `sBayes` models the prior for universal preference with a Dirichlet distribution:

$$P(\alpha_f) = \mathrm{Dir}(\psi_i) \qquad \text{for } i \in 1, \ldots N_f,$$

where N_f is the number of states for feature f and ψ_i is the concentration parameter for state i.
In contrast to universal preference and preference in a family, preference in an area is unknown a priori: contact areas and the features defining them are the result of an analysis with `sBayes` rather than a premise. Thus, the prior for preference in an area is necessarily "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 4: 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 5 shows an example of a cost matrix, again corresponding to data in *features.csv*.

Table 5: 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. Users can tune both the warm-up and the actual sampling.

Warm-up

For the warm-up, users can define the number of steps 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. The number of steps and chains depends on the complexity of the model, the number of languages, features, families and areas. If the number of iterations and chains are not provided by the user, `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 users can set the number of steps in the Markov chain, the sample size and the operators used in the MCMC. The number of steps 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 samples 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), the number uncorrelated, independent samples in the posterior.
The user can define how the MCMC proposes new samples, which operators are used and how often each operator is called. `sBayes` employs spatial operators to propose new areas – 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 to visualise and analyse 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, preference plots, weight plots, and DIC plots.

Maps

Maps show the posterior distribution of contact areas in geographic space. Maps include 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).

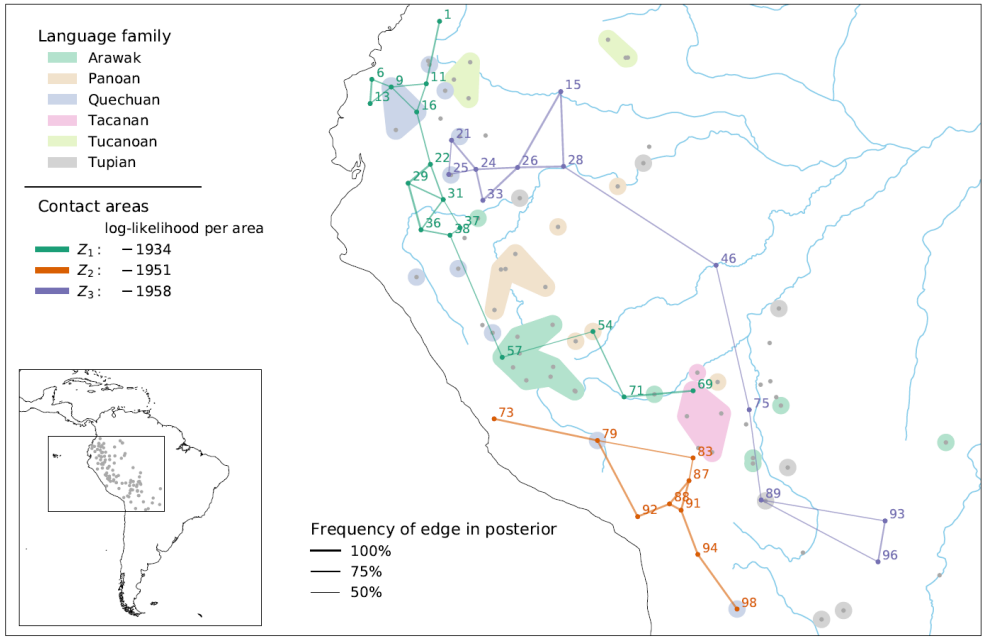

Figure 1: A map with three contact areas (green, orange and purple dots and lines).

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 2 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.

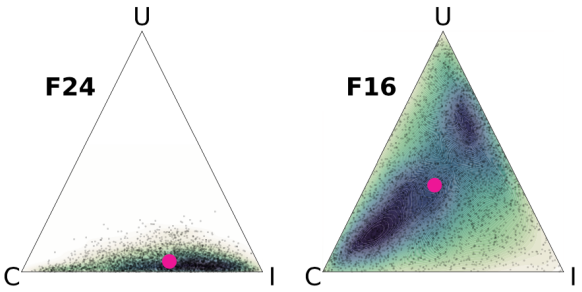

Figure 2: Weight plots for two features (F24, F16).

Preference plots

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 3), in a triangular probability simplex for three states (similar to the weights, see previous 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 3 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.

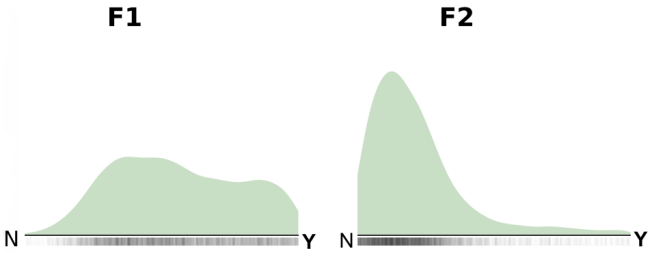

Figure 3: The density plot shows the posterior preference for two features (F1, F2) in an area

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, K, 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 – $K = 1$ to $K = 7$. The DIC levels off for $K = 4$, suggesting four salient contact areas in the data. As the DIC plot compares performance across models, it needs several result files as input.

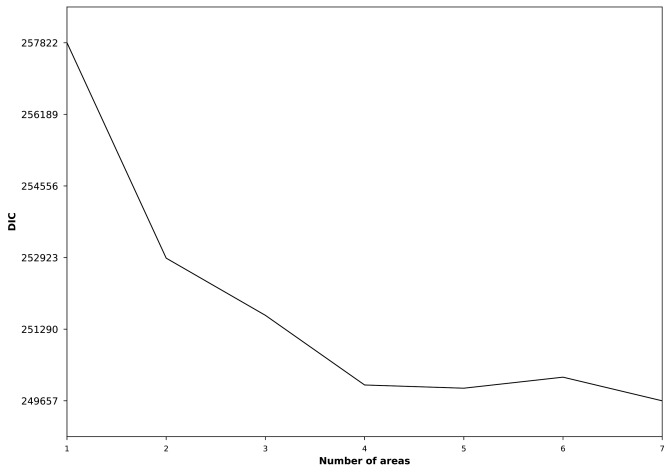

Figure 4: DIC plot for models with increasing number of areas ($K = 1$ to $K = 7$)

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 explores and inspects 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 keys:

- `data`: provides the 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 and the name of the result files

config.JSON: `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. In `projection`, users can provide a PROJ string or an EPSG code to define the geographic coordinate reference system (CRS) of the location data. If no CRS is provided, `sBayes` assumes that the location data are latitude/longitude data in WGS84 ("epsg:4326".)
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*). The location data are in ETRS89 Lambert Azimuthal Equal-Area projection ("epsg:3035").

```
"data": {
  "features" : "data/balkan_features.csv"
  "feature_states": "data/balkan_features_states.csv",
  "projection": "epsg:3035"}
```

Table 6 summarizes all keys in *data* and gives the default values and expected data types. *(required)* indicates that a key is mandatory and has to be set by the user.

Table 6: The *config.JSON* file: keys in `data`

key	data type	default value	description
`features`	string	(required)	file path to the *features.csv* file
`feature_states`	string	(required)	file path to the *feature_states.csv* file
`projection`	string	"epsg:4326"	CRS of the location data, as PROJ or EPSG

***config.JSON*: `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 (`areas`), and define the minimum and maximum number of languages in an area (`languages_per_area`). The key `prior` is discussed in detail below.

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,
  "areas": 5,
  "languages_per_area": {
     "min":10,
     "max": 40},
  "prior": {
    ...
  }
}
```

Table 7 summarizes all keys in `model` and gives the default values and expected data types.

Table 7: The *config.JSON* file: keys in `model`

key	**data type**	**default value**	
`inheritance`	boolean	(required)	Does the model consider inheritance?
`areas`	number	(required)	Number of contact areas in the model
`languages_per_area`	JSON	-	Minimum and maximum languages per area
`min`	number	3	Minimum number of languages per area
`max`	number	50	Maximum number of languages per area
`prior`	JSON		Defines the prior of the model. See below.

***config.JSON*: `model` > `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):

- `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 key takes as input a JSON object, which – depending on the type of prior – can itself have several sub-keys (for a full list see, Table 8).

`universal`

The prior for universal preference (`universal`) takes as input a JSON object with the keys `type`, `parameters`, and `file`. The default `type` is "uniform", which defines a uniform Dirichlet distribution, such that for each feature each applicable state is equally likely a priori.
If additional information is available to inform the prior, users can set `type` to "dirichlet" and provide the `parameters` of a Dirichlet prior distribution explicitly. Alternatively, users can provide the file path to an external JSON `file` (e.g. *prior_universal.JSON*) where the prior is parameterized. Externalizing the prior to a separate file is convenient when the model has many features and many states, in which case the parameterization of the prior distribution might make the *config.JSON* file overly convoluted. Moreover, section Constructing priors provides automated approaches to construct the prior from empirical data outside the study area. Both, `parameters` and the external JSON file have the exact same keys and values. In what follows, we explain how to parameterize the universal prior for the three features in Table 2 and the applicable states in Table 3.

In `parameters`, each feature is a key and all applicable states are sub-keys which take the concentration parameters of the Dirichlet distribution as values. Let us assume that the following three Dirichlet distributions define a weakly informative prior for universal preference for the features *pers_aff*, *role_mark*, and *vow_con* (Table 2):

$$P(\alpha_{pers\_aff}) = \text{Dir}(\psi_Y = 6.1, \psi_N = 7.9)$$

$$P(\alpha_{role\_mark}) = \text{Dir}(\psi_A = 2.0, \psi_B = 0.3, \psi_C = 4.7, \psi_D = 3.0, \psi_E = 4.0)$$

$$P(\alpha_{vow\_con}) = \text{Dir}(\psi_Y = 2.0, \psi_N = 12.0)$$

The JSON snippet below encodes the universal prior:

```
"prior": {
  ...
  "universal": {
    "type": "dirichlet",
    "parameters": {
      "pers_aff": {"Y": 6.1, "N": 7.9},
      "role_mark":{"A": 2.0, "B": 0.3,
                   "C": 4.7, "D": 3.0, "E": 4.0},
      "vow_con":{"Y": 2.0, "N": 12.0}
    }
  }
}
```

`inheritance`

The prior for preference in a family (`inheritance`) takes as input a JSON object for each family in *features.csv* (e.g. `Arawak`, `Tacanoan`, ...). Each family key has three sub-keys: `type`, `parameters`, and `file`. The default `type` is "uniform", which defines a uniform Dirichlet prior distribution: for each feature each applicable state is equally likely a priori. When there is additional information to inform the prior, users can set `type` to "dirichlet" and explicitly provide the `parameters` of the Dirichlet prior distribution in the family. Alternatively, users can provide the `file` path to an external JSON files (e.g. *prior_Arawak.JSON*) where the prior is parameterized. The `parameters` and the external JSON file have the exact same keys and values. In what follows, we explain how to parameterize the prior for Arawak for the three features in Table 2 and the applicable states in Table 3. For Tacanoan, we define a uniform prior.

In `parameters`, each feature is a key and all applicable states are sub-keys which take the concentration parameters of the Dirichlet distribution as values. Let us assume that the following three Dirichlet distributions define a weakly informative prior for universal preference in the Arawakan family for the features *pers_aff*, *role_mark*, and *vow_con* (Table 2):

$$P(\beta_{pers\_aff,Arawak}) = \text{Dir}(\psi_Y = 6.0, \psi_N = 1.0)$$

$$P(\beta_{role\_mark,Arawak}) = \text{Dir}(\psi_A = 2.0, \psi_B = 1.0, \psi_C = 2.9, \psi_D = 1.0, \psi_E = 0.1)$$

$$P(\beta_{vow\_con,Arawak}) = \text{Dir}(\psi_Y = 3.0, \psi_N = 4.0)$$

The JSON snippet below encodes the prior for Arawak and Tucanoan:

```
"prior": {
  ...
  "inheritance": {
    "Arawak" :{
      "type": "dirichlet",
      "parameters": {
        "pers_aff": {"Y": 6.0, "N": 1.0},
        "role_mark":{"A": 2.0, "B": 1.0,
                     "C": 2.9, "D": 1.0, "E": 0.1},
        "vow_con":{"Y": 3.0, "N": 4.0}}},
    "Tucanoan":{
      "type": "uniform"},
    ...
  }
}
```

`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 keys `type` and `parameters`. The default prior has type "uniform", in which case every spatial allocation of points in an area has the same prior probability. Other types are *Gaussian* and *cost_ based*.

For the Gaussian prior, the spatial locations in an area are evaluated against a two-dimensional Gaussian distribution. In `parameters`, the user provides a 2×2 variance-covariance matrix with units km^2 (`covariance`). The code snippet below defines a Gaussian geo-prior with a variance in x and y of 400 km^2 and a covariance of zero, i.e. the bivariate normal distribution is perfectly spherical.

```
"prior": {
   ...
   "geo": {
      "type": "gaussian",
      "parameters": {
          "covariance":[[400, 0],
                        [0, 400]]}
  }
}
```

For the cost-based geo prior, `sBayes` connects all locations in an area with a `linkage` criterion. The default criterion is "mst" which connects adjacent languages with a minimum spanning tree. Other linkage criteria are "delauney", which connects adjacent languages with a Delauney triangulation, and "complete" which connects every pair of distinct languages in an area. By default, costs are computed from the locations of the languages ("from_data"), either as Euclidean distance or as distance on a sphere (which of the two depends on the coordinate reference system of the input locations). Alternatively, users can provide the file path to a cost matrix to quantify the effort to travel between all pairs of languages.
The average costs necessary to link all languages in an area are then evaluated against an exponential decay function, for which the user provides an appropriate `rate`. The rate gives the mean expected costs between languages in contact. It defines how fast the probability for contact decreases with increasing costs in an area. The code snippet below enforces a cost-based geo prior. The costs are provided as travel times in hours in the file *travel_times.csv*. The rate is set to 12 hours and the minimum spanning tree is used as a linkage criterion.

```
"prior": {
    ...
    "geo": {
       "type": "cost_based",
       "parameters": {
         "costs": "travel_times.csv"
         "linkage": "mst",
         "rate": 12}
  }
}
```

`area_size`

There are two types of priors for area size: "uniform_area" 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 Priors). The following JSON snippet defines a prior that is uniform over size.

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

Table 8 summarizes all `prior` keys, gives the default values and expected data types.

Table 8: The *config.JSON* file: keys in `prior`

key	data type	default value	description
`universal`	JSON	-	the prior for universal preference
`type`	string	(required)	type of the prior, either "uniform" or "dirichlet"
`parameters`	JSON	-	parameterization of the prior distribution
`file`	string	-	alternatively: file path to *universal_prior.JSON*
`inheritance`	JSON	-	the prior for preference in a family
`family 1`	JSON	-	prior distribution for family 1
`type`	string	(required)	type of prior, either "uniform" or "dirichlet"
`parameters`	JSON	-	parameterization of the prior distribution for family 1
`file`	JSON	-	alternatively: file path to *prior_ <family 1>.JSON*
`family 2`	JSON	-	prior distribution for family 2
`type`	string	(required)	type of prior, either "uniform" or "dirichlet"
`parameters`	JSON	-	parameterization of the prior distribution for family 2
`file`	JSON	-	alternatively: file paths to *prior_ <family 2>.JSON*
...			
`contact`	JSON	-	the prior for preference in an area
`type`	string	"uniform"	only "uniform" priors are supported
`weights`	JSON	-	the weights prior
`type`	string	"uniform"	only "uniform" priors are supported
`geo`	JSON	-	the geo-prior
`type`	string	"uniform"	type of geo-prior: "uniform", "gaussian" or "cost_based"
`parameters`	JSON	-	additional parameters for defining the geo-prior
`covariance`	array	-	Gaussian covariance matrix (for "gaussian" geo-prior)
`costs`	string	"from_data"	"from_data" or file path to cost matrix (for "cost_based" geo-prior)
`rate`	number	-	rate of exponential distribution (for "cost_based" geo-prior)
`linkage`	string	"mst"	linkage criterion (for "cost_based" geo-prior)
`area_size`	JSON	-	the prior on area size.
`type`	string	"uniform_size"	type of prior, either "uniform_area" or "uniform_size".

***config.JSON*: `mcmc`**

In *mcmc* users define how `sBayes` samples from the posterior distribution. The key `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 (`warmup_chains`) and the number of steps in each chain (`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 (`steps`) and the number of samples retained from the chain `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 five 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": {
    ...
    "runs": 5,
    "steps": 1000000,
    "samples": 10000,
    "warmup": {
       "warmup_chains": 20,
       "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, the preference in a family or an area. `operators` 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 (`init_lang_per_area`) and define how often the proposal distribution for growing areas is spatially informed (`grow_to_adjacent`), 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 `init_lang_per_area` must be set such that the Markov chain is irreducible, which is only the case when the initial number of languages per area times the number of areas is smaller than the number of languages in the sample.

The following code snippet calls operators to modify areas (grow, shrink, swap) and universal preference 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": {
    ...
    "operators": {
        "area": 0.1,
        "weights": 0.3,
        "universal": 0.1,
        "inheritance": 0.2,
        "contact": 0.3},
    "grow_to_adjacent": 0.85,
    "init_lang_per_area": 6
}
```

Table 9 summarizes all `mcmcm` settings, gives the default values and expected data types. Note that the default values for `steps` and `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 9: The *config.JSON* file: keys in `mcmc`

key	data type	default value	description
`runs`	number	1	number of independent runs of the analysis
`steps`	number	100000	number of steps in the Markov chain
`samples`	number	1000	number of samples in the posterior
`warmup`	JSON	-	settings for the warmup
`warmup_chains`	number	15	number of warmup chains
`warmup_steps`	number	100000	number of warmup steps
`operators`	JSON	-	operators and their frequencies
`area`	number	0.05	frequency of area operator
`weights`	number	0.4	frequency of weights operator
`universal`	number	0.05	frequency of universal operator
`inheritance`	number	0.1	frequency of inheritance operator
`contact`	number	0.4	frequency of contact operator
`grow_to_adjacent`	number	0.85	frequency of grow steps to adjacent languages
`init_lang_per_area`	number	5	initial number of languages per area

***config.JSON*: `results`**

In `results` users provide the name and the file location of the results file. The key `path` gives the file path to the folder where the results will be saved. Users can give absolute or relative file paths. Relative paths are assumed to start from the location of the *config_plot.JSON* file. `log_file` creates a log file with meta information about the analysis, such as model parameters or acceptance//rejection statistics per operator.

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": {
   "path": "results",
   "log_file": true
}
```

Table 10 summarizes all keys in `results`, gives the default values and expected data types.

Table 10: The *config.JSON* file: keys in `results`

key	**data type**	**default value**	
`path`	string	"results"	file location to save the result
`log_file`	boolean	true	return a log file?

Inference

To runs `sBayes` from the command line, users simply call

```
sbayes <path_to_config_file>
```

`sBayes` runs the model, draws samples from the posterior and writes the results to two files, areas*.txt, for the posterior samples of the contact areas, and stats*.txt for the remaining parameters, including the (log-)likelihood, the prior and the posterior.

Post-processing

In post-processing, users can visualize and inspect the posterior distribution, either directly in `sBayes` or with third-party applications – such as `Tracer`. In the next section, we describe all custom plotting functions currently provided by `sBayes`.

Plotting

`sBayes` offers plotting functions to visualize the posterior distribution of the contact areas, the DIC, and other model parameters. Users provide the plotting parameters in the *config_plot.JSON* file. This file tells `sBayes` where to find the results and the original data, and which plots to generate.

The *config_plot.JSON* file

The *config_plot.JSON* file has the following keys:

- `results`: provides the file paths to the results of a `sBayes` analysis
- `data`: provides the file paths to the empirical data of the analysis
- `plots`: defines which plots to produce and how

config_plot.JSON: `results`

In `results`, the users provide the file paths to the results of a `sBayes` analysis (`path_in`) and the file paths to the output folder where the plots are saved (`path_out`). `in` has two sub-keys, `areas` for the posterior samples of the contact areas (areas*.txt), and `stats` for the posterior samples of the remaining parameters (stats*.txt). As shown below, users can provide several result files, which will be read in and plotted sequentially. The number of entries in `areas` and `stats` must be the same.
The following code snippet reads the results of three runs in `sBayes`for which the number of areas, K, was iteratively increased from $K = 1$ to $K = 3$. Once the plots are generated, they are saved in the folder "plots".

```
"results": {
  "path_in": {
    "areas": [
      "../results/K1/areas_K1_0.txt",
      "../results/K2/areas_K2_0.txt",
      "../results/K3/areas_K3_0.txt"],
    "stats": [
      "../results/K1/stats_K1_0.txt",
      "../results/K2/stats_K2_0.txt",
      "../results/K3/stats_K3_0.txt"]},
  "path_out": "plots"
```

Table 11 shows all keys in *config_plot.JSON* > *results* and gives default values and expected data types.

Table 11: The *config_plot.JSON* file: keys in `results`

key	**data type**	**default value**	**description**
`path_in`	JSON	-	file path to the results
`areas`	array	(required)	file path to one or several areas*.txt files
`stats`	array	(required)	file path to one ore several stats*.txt files
`path_out`	string	"plots"	file path to the output folder for saving the plots

config_plot.JSON: `data`

The key `data` points to the empirical data which were used as an input for the `sBayes` analysis. Users provide the file paths to the *features.csv* file (`features`), the applicable states for all features (`feature_states`), and the coordinate reference system of the location data (`projection`). If no CRS is provided, `sBayes` assumes that the location data are latitude/longitude data in WGS84 ("epsg:4326"). The following JSON snippet tells plotting 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_plot.JSON*). The location data are in ETRS89 Lambert Azimuthal Equal-Area projection ("epsg:3035").

```
"data": {
  "features" : "data/balkan_features.csv"
  "feature_states": "data/balkan_features_states.csv",
  "projection": "epsg:3035"}
```

Table 12 shows all keys in *config_plot.JSON* > *data* and gives default values and expected data types.

Table 12: The *config_plot.JSON* file: keys in `data`

key	data type	default value	description
`features`	string	(required)	path to the *features.csv* file
`feature_states`	string	(required)	path to to the *feature_states.csv* file
`projection`	string	"epsg:4326"	CRS of the location data, as PROJ or EPSG

***config_plot.JSON*: `plots`**

In `plots`, users define which plots to make and how to make them. `plots` has the following sub-keys

- `map`: creates a map of the posterior contact areas
- `weights_plot`: creates plots of the posterior weights
- `preference_plot`: creates plots of the posterior preference of each state (universally, in each family and in each area)
- `dic_plot`: visualizes the deviance information criterion across several models

Table 13 shows all keys in *config_plot.JSON* > *plot* and gives default values and expected data types.

Table 13: The *config_plot.JSON* file: keys in `plot`

key	**data type**	**default value**	**description**
`map`	JSON	*	creates a map of the posterior contact areas
`weight_plot`	JSON	*	creates plots of the posterior weights
`preference_plot`	JSON	*	creates plots of the posterior preference of each state (universally, in each family and in each area)
`dic_plot`	JSON	*	visualizes the deviance information criterion across several models

* see sub-keys

config_plot.JSON: `plots` > `map`

`map` creates a map of the posterior distribution of contact areas. Maps visualize the spatial locations of all languages, their assignment to contact areas and, optionally, the language families. Users have a wide array of options to customize the map to their liking. They can change the content of the map (`content`), either visualizing the full posterior density or a consensus map. They can change the appearance of individual map items (`graphic`), tweak the projection and add base map items (`geo`). Finally, users can add a legend and an overview map (`legend`) and define the output format `output`. `map` has the following sub-keys, all of which are JSON objects.

- `content`: defines the map content
- `geo`: adds a base map, changes the map extent and the map projection
- `graphic`: changes the appearance of individual map items
- `legend`: adds a legend and/or an overview map
- `output`: defines the output format, size and resolution

***config_plot.JSON*: `plots` > `map` > `content`**

The key `content` defines the map type and all map items. There are two different map types (`type`): users can either visualize the full posterior density of all contact areas (`"density_map"`) or a high-level summary of the posterior, the consensus areas (`"consensus_map"`). The density map shows all languages that are in the posterior. Line width indicates how often two languages appear together in the same area in the posterior. For consensus areas, the map is generalized. `"min_posterior_frequency"` defines the degree of generalization. For example, a `"min_posterior_frequency"` of 1 retains only those language which are in all posterior contact areas, while a value of 0.3 retains languages which appear in at least 30% of the posterior, preserving more of the posterior uncertainty. After generalization, all remaining languages are connected with a Gabriel graph.
With `plot_families` users can visualize the language families, `burn_in` specifies which part of the posterior is discarded as burn-in.
The following code snippet creates a consensus map showing languages which are at least in 75% of the posterior contact areas. The map visualizes all language families and uses a shorthand labels the languages. The first 10% of all samples are discarded as burn-in.

```
"map": {
  ...
  "content": {
    "type" : "consensus_map",
    "min_posterior_frequency": 0.75,
    "plot_families": true,
    "burn_in": 0.1}
}
```

Table 14 shows all keys in *config_plot.JSON* > *map* > *content* and gives default values and expected data types.

Table 14: The *config_plot.JSON* file: keys in `map > content`

key	data type	default value	description
`type`	string	consensus_map	type of plot, "density_map" or "consensus_map"
`min_posterior_frequency`	number	0.5	degree of generalization (only for consensus maps)
`plot_families`	boolean	true	Plot the language families?
`burn_in`	number	0.2	fraction of posterior samples discarded as burn-in

***config_plot.JSON*: `plots` > `map` > `geo`**

In `geo`, users can set the cartographic projection of the map, add base maps and define the map extent. In `map_projection` users provide either a PROJ string or an EPSG code to define the coordinate reference system (CRS) of the map. If no CRS is provided, `sBayes` uses the CRS of the input data. In `base_map` users provide the file paths to polygon and line geometries which are displayed as base maps. `base_map` has three sub-keys: `geojson_polygon`, which expects a file path to a GeoJSON polygon geometry, `geojson_line`, which expects a file path to a GeoJSON line geometry, and `add`, which adds the base map when set to true and omits it if set to false (alternatively, the `base_map` key can simply be left empty). Finally, in `map_extent` users give the map extent in the x-direction (`x`) and y-direction (`y`) in the CRS of the output map. If no extent is provided, the plotting function uses the bounding box of the data plus an offset as map extent.
The following code snippet plots the map in the CRS of the World Geodetic System 1984, adds a land mask and a river network as a base map, and sets the map extent to cover Central Europe.

```
"map": {
  ...
    "geo": {
      "map_projection": "EPSG:4326",
      "base_map": {
        "add": true,
        "geojson_polygon": "../data/map/land.geojson",
        "geojson_line": "../data/map/rivers_lake.geojson"},
      "extent": {
        "x": [5, 22],
        "y": [45, 55]}
}
```

Table 15 summarises all keys in *config_plot.JSON* > *map* > *geo* and gives default values and expected data types.

Table 15: The *config_plot.JSON* file: keys in `map> geo`

key	**data type**	**default value**	**description**
`map_projection`	string	*	PROJ4 string or EPSG code to define a CRS
`base_map`	JSON	-	Settings for the base map
`add`	boolean	false	Add the base map?
`geojson_polygon`	string	-	File path to GeoJSON polygon geometry
`geojson_line`	string	-	File path to GeoJSON line geometry
`extent`	JSON	-	Defines the map extent
`x`	array	*	Range of the extent in x-direction
`y`	array	*	Range of the extent in y-direction

* default value is derived from input data

***config_plot.JSON*: `plots` > `map` > `graphic`**

In `graphic` users can change the appearance of the individual map items, including the languages (`languages`), the posterior areas (`areas`), the families (`families`) and the base map (`base_map`). Alternatively, users can leave all parameters untouched, in which case the default design is used. The key (`languages`) tunes how the languages are displayed on the map: the sub-key `size` changes the size of the point markers, `color` changes their color, and (`labels`) adds labels. For reasons of readability, on the map labels are shown as numeric shorthand notations, for which users can output a correspondence table (see key (`output`)). The (`areas`) key has four sub-keys: users can change the size of the point markers (`size`), set the color of each contact area (`color`), and change the `width` and the transparency (`alpha`) of the lines connecting the languages in an area. In `families`, users can change the `color` and the `shape` of the family polygons. For a small shape value, each language is assigned to their own circular polygon. Users can set the size of this polygon (`size`). When the shape value is increased, the family polygons are more and more generalized, such that neighbouring languages from the same family are aggregated to larger polygons using alpha shapes. These polygons have a `buffer` (in map units). For good results without overlapping polygons expect to tweak the shape and the buffer values iteratively. The key `base_map` defines the appearance of the base map polygons (`polygon` – with sub-keys `color`, `outline_color`, and `outline_width`, and the base map lines (`line` – with sub-keys `color` and `width`). The following code snippet tweaks the appearance of the languages, the three areas, the two families and the base map:

```
"map": {
  ...
    "graphic": {
      "languages": {
        "size": 10,
        "color": "grey",
        "label": true},
      "areas": {
        "size" : 15,
        "color": ["green", "red", "blue"],
        "width": 2,
        "alpha": 1},
      "families": {
        "size": 50,
        "color": ["lightyellow", "lavender"]
        "buffer": 1,
        "shape": 0.1}
      "base_map": {
        "polygon": {
          "color": "orange",
          "outline_color": "black",
          "outline_width": 1},
        "line": {
          "color": "blue",
          "width": 2}}
}
```

Table 16 summarises all keys in *config_plot.JSON* > *map* > *graphic* and gives default values and expected data types.

Table 16: The *config_plot.JSON* file: keys in `map > graphic`

key	**data type**	**default value**	**description**
`languages`	JSON	-	appearance of languages on the map
`size`	number	15	size of the point markers for languages
`color`	string	"grey"	color of the point marker for languages
`label`	boolean	true	Add labels for the languages
`areas`	JSON	-	appearance of areas on the map
`size`	number	20	size of the point markers for languages in an area
`color`	array	*	color of each contact area
`width`	number	2	width of the line between languages in an area
`alpha`	number	0.5	transparency of the line between languages in an area
`families`	JSON		appearance of the families on the map
`size`	number	200	Size of a single language polygon
`color`	array	*	color of each family
`buffer`	array	0.3	buffer around the generalized language polygons
`shape`	array	1	defines the level of generalization for family polygons
`base_map`	JSON	-	appearance of the base map
`polygon`	JSON	-	the appearance of the polygon geometries
`color`	string	"white	face color of the polygons
`outline_color`	string	"grey"	outline color of the polygons
`outline_width`	number	0.5	outline width of the polygons
`line`	JSON	-	the appearance of the line geometries
`color`	string	"skyblue"	color of the lines
`width`	number	1	width of the lines

* A pleasing assortment of different colors. Let yourself be surprised!

***config_plot.JSON*: `plots` > `map` > `legend`**

In `legend` users can add legend items to the map or remove them. When no `legend` item is provided the default legend is used. There a five legend items: a legend for the posterior contact areas (`areas`), a legend for the lines in a contact area (`lines`), a legend for the families (`families`), a correspondence table (`correspondence`) and an overview map (`overview`). For each of these, users can decide whether or not to add the item (`add`) and where to place it on the map canvas (`position`) (except for the correspondence table will always be placed below the map). The position is given for the lower-left corner of the legend item relative to the lower-left corner of the image and the map extant. For example, $[0.1, 0.3]$ means that the legend item is located 10% of the map width to the right and 30% of the map height up, relative to the lower-left corner of the map.

The `areas` legend lists all contact areas and ranks them according to the relative posterior probability. Additionally, users can show the log-likelihood per area (`log_likelihood`). The `lines` legend helps interpreting line thickness in the map (the thicker the line the more often two languages appear together in the same area). The legend shows three reference lines with decreasing thickness together with the corresponding posterior frequency. Users can set the frequency of each reference lines (`reference_frequency`). For example, a frequency of 0.2 means that the thickness of the reference line corresponds to two languages that together appear in the posterior 20%. The `families` legend explains the families on the map. The legend can only be added when families are shown. The `correspondence` table links the labels in numeric short hand notation on the map to the names of the actual languages. Entries in the correspondence table can be colored to match the colors of the contact areas (`color_labels`). Coloring is only available for consensus maps. Moreover, users can change the number of columns (`n_columns`) in the correspondence table, the font size for the entries (`font_size`) and the height of the table (`table_height`). Finally, `overview` adds an overview map. Users can set the extent(`extent`) in x-direction (`x`) and y-direction (`y`) and give the size of the overview image on the map in centimetres (`height`, (`width`).

The following code snippet adds a legend item for `areas`, changes its position on the map and shows the log-likelihood per area. The code adds a legend item for `lines` and sets the the three reference lines to 20%, 60% and 80% posterior frequency. Finally, it removes the legend item for `families`. It adds a correspondence table with colored entries and a $4x5$ cm overview map of (roughly) Europe.

```
"map": {
  ...
    "legend": {
      "areas": {
        "add": true,
        "position": [0.1, 0.1],
        "log-likelihood": true},
      "lines": {
        "add": true,
        "reference_frequency}" = [0.2, 0.6, 0.8],
      "families": {
        "add": false}
      "correspondence": {
        "add" = true,
        "color_labels" = false},
      "overview":{
        "add" = true,
        "height" = 4
        "width" = 5,
        'extent": {
            x = [-10, 50],
            y = [35, 70]
        }
    }
}
```

Table 17 summarises all keys in *config_plot.JSON* > *map* > *legend* and gives default values and expected data types.

Table 17: The *config_plot.JSON* file: keys in `map > legend`

key	data type	default value	description
`areas`	JSON	-	legend for areas
`add`	boolean	true	add legend entry for areas?
`position`	array	[0.01, 0.71]	relative position of the legend entry
`log-likelihood`	boolean	false	show log-likelihood per area?
`lines`	JSON	-	legend for lines
`add`	boolean	false	add legend entry for lines?
`position`	array	[0.4, 0.15]	relative position of the legend entry
`reference_frequency`	array	[1, 0.75, 0.5]	line thickness of the three reference lines
`families`	JSON	-	legend for families
`add`	boolean	false	add legend entry for families?
`position`	array	[0.01, 0.98]	relative position of the legend entry
`correspondence`	JSON	-	the correspondence table
`add`	boolean	false	add correspondence table
`n_columns`	number	4	number of columns in the correspondence table
`font_size`	number	12	font size in the correspondence table
`table_height`	number	0.2	height of the correspondence table
`color_labels`	boolean	false	color the labels in the table to match contact areas?
`overview`	JSON	-	overview map
`add`	boolean	false	add the overview map?
`position`	array	[0.62, 0.6]	relative position of the overview map
`width`	number	4	width of the overview map (in cm)
`height`	number	4	height of the overview map (in cm)
`extent`	JSON	-	extent of the overview map
`x`	array	*	range of the extent in x-direction
`y`	array	*	range of the extent in y-direction

* default value is derived from input data

***config_plot.JSON*: `plots` > `map` > `output`**

In `output`, users define the `width` and the `height` of the output figure, its file `format` and `resolution`. The following code snippet creates a figure of size $15x10$cm in png-format with a resolution of 400 pixels per inch.

```
"map": {
  ...
    "output": {
      "width": 15,
      "height": 10,
      "format": "png",
      "resolution": 400}
}
```

Table 18 summarises all keys in *config_plot.JSON* > *map* > *output* and gives default values and expected data types.

Table 18: The *config_plot.JSON* file: keys in `map > output`

key	**data type**	**default value**	**description**
`width`	number	20	width of the output figure in cm
`height`	number	10	height of the output figure in cm
`format`	string	"pdf"	file format of the output figure
`resolution`	number	300	resolution of the output figure in pixels per inch

config_plot.JSON: `plots` > `weight_plot`

Weight plots visualize the posterior densities of the weights per feature: how well does each of the effects – universal preference, inheritance and contact – explain 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, the right lower corner the weight for inheritance, and the upper corner the weight for universal preference. The weight plots of several features are combined into one overall figure. The plotting function has three sub-keys: `content`, `graphic`, and `output`.

config_plot.JSON: `plots` > `weight_plot` > `content`

In `content`, users specify for which of the features the weights are plotted (`features`). Users pass the relevant features in an array. For example, [3] generates a weight plot for feature three only; [3, 4, 17] generates weight plots for features three, four and seventeen. When the array is left empty or `features` are not provided explicitly, the plotting function generates weight plots for all features. Moreover, users can specify in `burn_in` which part of the posterior to discard as burn-in.
The following code snippet creates weight plots for features 5,6, and 8. The first 10% of the posterior samples are discarded as burn-in.

```
"weight_plot": {
  ...
    "content": {
      "features": [5, 6, 8],
      "burn_in": 0.1}
}
```

Table 19 summarises all keys in *config_plot.JSON* > *weight_plot* > *content* and gives default values and expected data types.

Table 19: The *config_plot.JSON* file: keys in `weight_plot > content`

key	data type	default value	description
`features`	array	[ ]	features for which weight plots area created
`burn_in`	number	0.2	fraction of the posterior samples discarded as burn-in

***config_plot.JSON*: `plots` > `weight_plot` > `graphic`**

In `graphic`, users can label of each of the corners in the triangular probability simplex (`labels`. The default labels are ["U", "C", "I"]. The sub-key `title` adds the feature name as a title for each weight plot. Moreover, users can specify how the sub-plots for single features are combined in the overall figure: `n_columns` gives the number of columns in the overall figure.
The following code snippet uses alternative labels and removes the title for each sub-plot. All sub-plots are combined in an overall figure with five columns.

```
"weight_plot": {
  ...
    "graphic": {
      "labels": ["Universal", "Contact", "Inheritance"],
      "title": false,
      "n_columns": 5}
}
```

Table 20 summarises all keys in *config_plot.JSON* > *weight_plot* > *graphic* and gives default values and expected data types.

Table 20: The *config_plot.JSON* file: keys in `weight_plot > graphic`

key	data type	default value	description
`labels`	array	["U", "C", "I"]	labels for the corners of the probability simplex
`title`	boolean	true	Use the feature name as a title for each sub-plot?
`n_columns`	number	5	number of columns in the overall plot

***config_plot.JSON*: `plots` > `weight_plot` > `output`**

In `output`, users define the width and the height of each of the sub-plots (`width_subplot`, `height_subplot`, the file `format` of the output file and the `resolution`.
The following code snippet creates $4x4$ cm sub-plots. All sub-plots are combined in an overall figure and saved in png-format with a resolution of 400 pixels per inch.

```
"weight_plot": {
  ...
    "output": {
      "width_subplot": 4,
      "height_subplot": 4,
      "format": "png",
      "resolution": 400}
}
```

Table 21 summarises all keys in *config_plot.JSON* > *weight_plot* > *output* and gives default values and expected data types.

Table 21: The *config_plot.JSON* file: keys in `weight_plot > output`

key	data type	default value	description
`width_subplot`	number	3	width of each subplot in cm
`height_subplot`	number	3	height of each subplot in cm
`format`	string	"pdf"	file format of the output figure
`resolution`	resolution	300	resolution of the output figure in pixels per inch

config_plot.JSON: `plots` > `preference_plot`

The preference plots visualize the posterior preference for each of the states of a feature, either universally, in a family or in 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 3), in a triangular probability simplex for three states, a square for four states, a pentagon for five, and so on. `preference_plot` combines the sub-plot for several features (per family, per area or globally) in a single figure.

config_plot.JSON: `plots` > `preference_plot` > `content`

In `content`, users specify for which of the `features` preferences are plotted. Users pass the relevant features in an array. For example, [3] generates preference plots for feature three only; [3, 4, 17] generates preference plots for features three, four and seventeen. When the array is left empty or `features` are not provided explicitly, the plotting function generates plots for all features. Moreover, users can define which `preference` to plot, i.e. preference in one of the areas, in one of the families, or universal preference. For example, ["universal", "area_1", "Arawak"] creates three plots: a plot for universal preference, for preference in area 1, and for preference in the Arawakan family. General, all sub-plots for one preference are combined in a single figure. When preference is not provided explicitly, preference plots are generated for all the areas as default. Finally, users can specify in `burn_in` which part of the posterior to discard as burn-in.
The following code snippet creates sub-plots for features 7, 8, and 9 and preference in areas 2 and 3. The first 10% of the posterior samples are discarded as burn-in.

```
"preference_plot": {
  ...
    "content": {
      "features": [7,8,9],
      "preference": ["area_2", "area_3"]
      "burn_in": 0.1}
}
```

Table 22 summarises all keys in *config_plot.JSON* > *preference_plot* > *content* and gives default values and expected data types.

Table 22: The *config_plot.JSON* file: keys in `preference_plot > content`

key	data type	default value	description
`features`	array	[ ]	features for which preference plots area created
`preference`	array	all areas	preference (area, family, global) for which plots area created
`burn_in`	number	0.2	fraction of the posterior samples discarded as burn-in

***config_plot.JSON*: `plots` > `preference_plot` > `graphic`**

In `graphic`, users can change the appearance of the preference plot. Specifically, they can use the state names of the features to label both ends of the ridge, or the corners of the probability simplex in case there are more than two states (`labels`). The sub-key `title` adds the feature name as a title for each preference plot. Moreover, users can specify how the sub-plots for single features are combined in the overall figure: `n_columns` gives the number of columns in the overall figure.
The following code snippet uses the state names as labels and the features names as a title for each sub-plot. Sub-plots are combined in overall figures with six columns.

```
"preference_plot": {
  ...
    "graphic": {
      "labels": true,
      "title": true,
      "n_columns": 6}
}
```

Table 23 summarises all keys in *config_plot.JSON* > *preference_plot* > *graphic* and gives default values and expected data types.

Table 23: The *config_plot.JSON* file: keys in `preference_plot > graphic`

key	data type	default value	description
`labels`	boolean	true	use the state names as labels for the corners of the probability simplex?
`title`	boolean	true	Use the feature name as a title for each sub-plot?
`n_columns`	number	5	number of columns in the overall plot

***config_plot.JSON*: `plots` > `preference_plot` > `output`**

In `output`, users define the width and the height of each of the sub-plots (`width_subplot`, `height_subplot`, the file `format` of the output file and the `resolution`.
The following code snippet creates $4x4$ cm sub-plots. All sub-plots are combined in an overall figure and saved in png-format with a resolution of 400 pixels per inch.

```
"weight_plot": {
  ...
    "output": {
      "width_subplot": 4,
      "height_subplot": 4,
      "format": "png",
      "resolution": 400}
}
```

Table 24 summarises all keys in *config_plot.JSON* > *preference_plot* > *output* and gives default values and expected data types.

Table 24: The *config_plot.JSON* file: keys in `preference_plot > output`

key	data type	default value	description
`width_subplot`	number	3	width of each subplot in cm
`height_subplot`	number	3	height of each subplot in cm
`format`	string	"pdf"	file format of the output figure
`resolution`	resolution	300	resolution of the output figure in pixels per inch

config_plot.JSON: `plots` > `dic_plot`

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, K. Plotting the DIC is only meaningful when several areas*.txt and stats*.txt files from different models are provided in `results`.

config_plot.JSON: `plots` > `dic_plot` > `content`

In `content`, users can change which models appear in the DIC and how they are ordered (`models`). Typically, models are ordered from simplest to most complex, e.g. from fewest to most areas. When no models are provided by the user, the DIC is plotted for all models from `results`. Moreover, users can specify in `burn_in` which part of the posterior to discard as burn-in.
The following code snippet creates a DIC plot for models 1, 3, and 5 in results. The first 30% of the posterior samples of each model are discarded as burn-in.

```
"dic_plot": {
  ...
    "content": {
      "models": [1,3,5],
      "burn_in": 0.3}
}
```

Table 22 summarises all keys in *config_plot.JSON* > *dic_plot* > *content* and gives default values and expected data types.

Table 25: The *config_plot.JSON* file: keys in `dic_plot > content`

key	data type	default value	description
`models`	array	all models	models that appear in the DIC plot
`burn_in`	number	0.2	fraction of the posterior samples discarded as burn-in

***config_plot.JSON*: `plots` > `dic_plot` > `graphic`**

In `graphic`, users can change the appearance of the DIC plot. Specifically, they can label the models in the plot (`labels`) and change whether consecutive DIC values are connected with a line (`line_plot`) or a scatter plot is used. As a default, the models are labelled in ascending order, starting from 1.
The following code uses the labels "K1", "K3", and "K5". Consecutive DIC values are vconnected with a line.

```
"dic_plot": {
  ...
    "graphic": {
      "labels": ["K1", "K3", "K5"],
      "line_plot": true
}
```

Table 26 summarises all keys in *config_plot.JSON* > *dic_plot* > *graphic* and gives default values and expected data types.

Table 26: The *config_plot.JSON* file: keys in `preference_plot > graphic`

key	**data type**	**default value**	**description**
`labels`	array	*	labels of the models
`line_plot`	boolean	true	Connect consecutive DIC values with a line?

* default value is derived from input data

***config_plot.JSON*: `plots` > `dic_plot` > `output`**

In `output`, users define the width and the height of the DIC plot (`width`, `height`, the file `format` of the output file and the `resolution`.
The following code snippet creates a $10x8$ cm DIC plot in png-format with a resolution of 200 pixels per inch.

```
"dic_plot": {
  ...
    "output": {
      "width": 10,
      "height": 8,
      "format": "png",
      "resolution": 200}
}
```

Table 27 summarises all keys in *config_plot.JSON* > *dic_plot* > *output* and gives default values and expected data types.

Table 27: The *config_plot.JSON* file: keys in `dic_plot > output`

key	data type	default value	description
`width`	number	9	width of the plot in cm
`height`	number	6	height of the plot in cm
`format`	string	"pdf"	file format of the output figure
`format`	resolution	300	resolution of the output figure in pixels per inch