scikit-neuromsi

Release 0.0.1

Paredes, Renato; Cabral, Juan

Mar 29, 2022

CONTENTS:

Scikit-neuromsi is an open-source Python framework that simplifies the implementation of neurocomputational models of multisensory integration.

CHAPTER

ONE

MOTIVATION

Research on the the neural process by which unisensory signals are combined to form a significantly different multisensory response has grown exponentially in the recent years. Nevertheless, there is as yet no unified theoretical approach to multisensory integration. We believe that building a framework for multisensory integration modelling would greatly contribute to originate a unifying theory that narrows the gap between neural and behavioural multisensory responses.

Authors
Renato Paredes (E-mail: paredesrenato92@gmail.com)

CHAPTER

TWO

REPOSITORY AND ISSUES

https://github.com/renatoparedes/scikit-neuromsi

2.1 Installation

This is the recommended way to install scikit-neuromsi.

2.1.1 Installing with pip

Make sure that the Python interpreter can load scikit-neuromsi code. The most convenient way to do this is to use virtualenv, virtualenvwrapper, and pip.

After setting up and activating the virtualenv, run the following command:

```
$ pip install scikit-neuromsi
...
```

That should be enough.

2.1.2 Installing the development version

If you'd like to be able to update your scikit-neuromsi code occasionally with the latest bug fixes and improvements, follow these instructions:

Make sure that you have Git installed and that you can run its commands from a shell. (Enter *git help* at a shell prompt to test this.)

Check out scikit-neuromsi main development branch like so:

```
$ git clone https://github.com/renatoparedes/scikit-neuromsi.git
...
```

This will create a directory *scikit-neuromsi* in your current directory.

Then you can proceed to install with the commands

```
$ cd scikit-neuromsi
$ pip install -e .
...
```

2.2 Scikit-neuromsi Tutorial

This tutorial is intended to serve as a guide for using `scikit-neuromsi` to implement neurocomputational models of multisensory integration.

2.2.1 General imports

The first thing we will do is import the necessary libraries. In general you will need the following:

- skneuromsi (*skneuromsi*) is the library that we present in this tutorial.
- numpy (*numpy*) this library will allow you to perform numerical computations.
- matplotlib (*matplotlib.pyplot*) this librarly will allow you to visualise your results.

Note: In this tutorial we assume that you already have a basic knowledge of `numpy` and `matplotlib` for scientific computing.

[1]:
```
import skneuromsi
import numpy as np
import matplotlib.pyplot as plt
```

2.2.2 Implementation of the Alais and Burr bimodal integration model

To easily implement the model developed by Alais and Burr (2004) you can import the corresponding module and instatiate the `AlaisBurr2004` class:

[2]:
```
from skneuromsi.alais_burr2004 import AlaisBurr2004

model = AlaisBurr2004()
model
```

[2]:
```
AlaisBurr2004(possible_locations=array([-20.  , -19.99, -19.98, ...,  19.97,  19.98,  19.
↪99]), auditory_sigma=3.0, visual_sigma=3.0)
```

By calling the model you can explore its main built-in parameters: - `possible_locations`: All the possible locations where the stimulus could be delivered (in degrees). - `auditory_sigma`: Standard deviation of the auditory estimate. - `visual_sigma`: Standard deviation of the visual estimate.

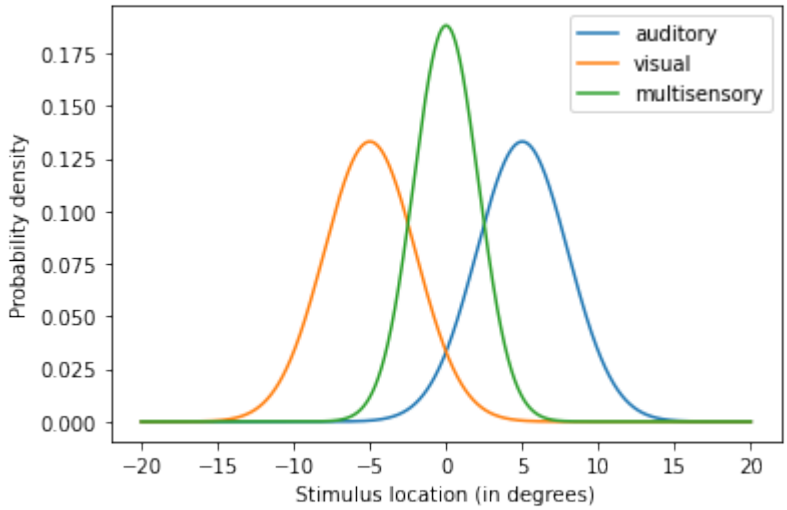
Now let's run the model for equidistant auditory and visual locations:

[3]:
```
res = model.run(visual_location=-5, auditory_location=5)
res
```

[3]:
```
{'auditory': array([1.10692781e-16, 1.13810052e-16, 1.17013810e-16, ...,
        5.20955703e-07, 5.12359295e-07, 5.03899139e-07]),
 'visual': array([4.95573172e-07, 5.03899139e-07, 5.12359295e-07, ...,
        1.20306417e-16, 1.17013810e-16, 1.13810052e-16]),
 'multisensory': array([9.38267850e-21, 9.80898311e-21, 1.02544291e-20, ...,
        1.07198655e-20, 1.02544291e-20, 9.80898311e-21])}
```

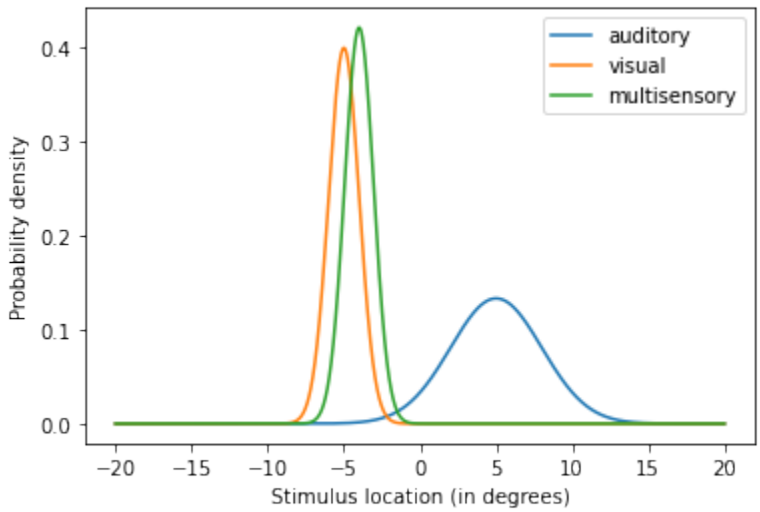
The model outputs three `numpy.arrays` containing the results of both unisensory estimators and the multisensory estimator. To make sense of our results, let's visualise the output:

[4]:
```
for k in res:
    plt.plot(model.possible_locations, res[k], label=k)
plt.legend()
plt.xlabel("Stimulus location (in degrees)")
plt.ylabel("Probability density")
plt.show()
```



The plot shows how both auditory and visual estimates are combined into a single multisensory estimate. Now let's try a different configuration of the model:

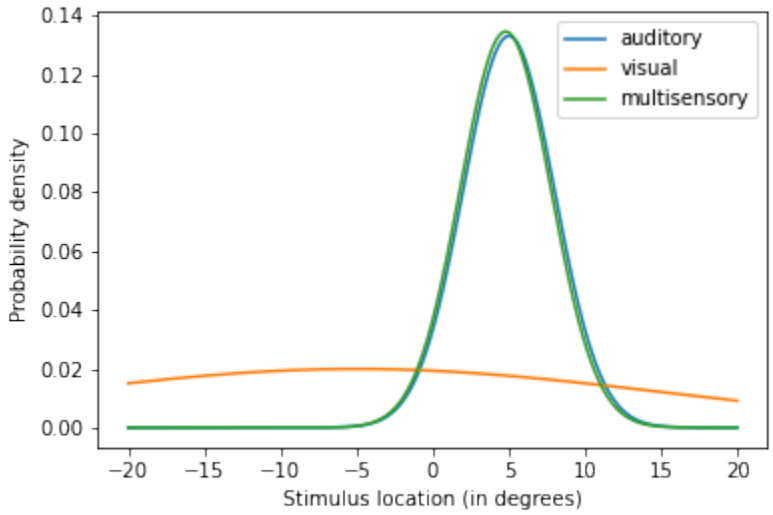
[5]:
```
alter_model = AlaisBurr2004(visual_sigma=1, auditory_sigma=3)
alter_res = alter_model.run(visual_location=-5, auditory_location=5)

for k in res:
    plt.plot(alter_model.possible_locations, alter_res[k], label=k)
plt.legend()
plt.xlabel("Stimulus location (in degrees)")
plt.ylabel("Probability density")
plt.show()
```



In this new configuration we increased the precision of the visual estimate. By doing so, the multisensory estimate of the stimulus location is dramatically biased towards the visual estimate.

The opposite happens if we decrease the visual stimulus precision:

[6]:
```
alter_model = AlaisBurr2004(visual_sigma=20, auditory_sigma=3)
alter_res = alter_model.run(visual_location=-5, auditory_location=5)

for k in res:
    plt.plot(alter_model.possible_locations, alter_res[k], label=k)
plt.legend()
plt.xlabel("Stimulus location (in degrees)")
plt.ylabel("Probability density")
plt.show()
```



By manipulating the precision of the unisensory estimates you've explored computationally the principles of the MLE estimation behind the model. Refer to the documentation for further information about parameters to manipulate.

> This demonstration of the Alais and Burr model mechanics is inspired in the Computational Cognitive Neuroscience course materials developed by Dr. Peggy Series at The University of Edinburgh.

2.2.3 Implementation of the Ernst and Banks visual-haptic integrator model

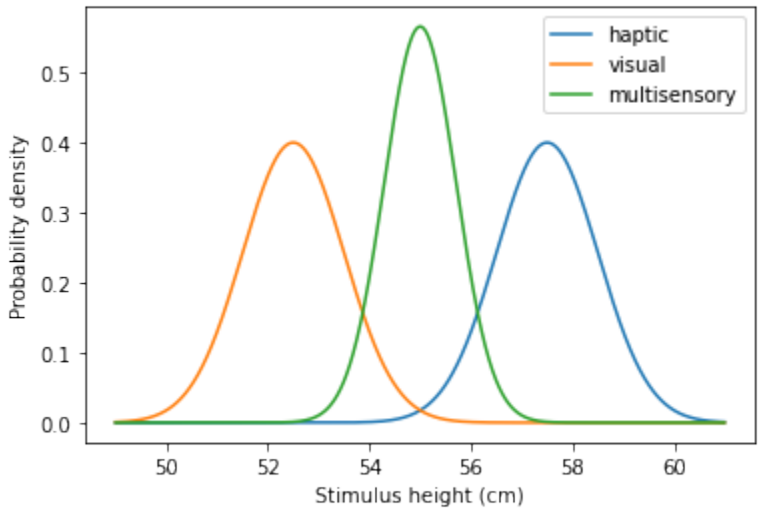
As in the previous section, you can implement the model developed by Ernst and Banks (2002) by importing the corresponding module and instantiating the `ErnstBanks2002` class:

[7]:
```
from skneuromsi.ernst_banks2002 import ErnstBanks2002

model = ErnstBanks2002()
model
```

[7]:
```
ErnstBanks2002(possible_heights=array([49.  , 49.01, 49.02, ..., 60.97, 60.98, 60.99]),␣
↪haptic_sigma=1, visual_sigma=1)
```

Let's run the model for two conflicting stimulus heights:

[8]:
```
model = ErnstBanks2002()
res = model.run(visual_height=52.5, haptic_height=57.5)

for k in res:
    plt.plot(model.possible_heights, res[k], label=k)
plt.legend()
plt.xlabel("Stimulus height (cm)")
plt.ylabel("Probability density")
plt.show()
```



The model mechanics are the same as in the previous section. We'll leave its exploration to you!

2.2.4 Implementation of the Kording Bayesian Causal Inference model

You can implement the model Causal Inference model developed by Kording et al (2002) by importing the corresponding module and instantiating the `Kording2007` class:

[12]:
```
from skneuromsi.kording2007 import Kording2007
import matplotlib.pyplot as plt

model = Kording2007()
model
```

[12]:
```
Kording2007(possible_locations=(array([-42.        , -40.28571429, -38.57142857, -36.
↪85714286,
       -35.14285714, -33.42857143, -31.71428571, -30.        ,
       -28.28571429, -26.57142857, -24.85714286, -23.14285714,
       -21.42857143, -19.71428571, -18.        , -16.28571429,
       -14.57142857, -12.85714286, -11.14285714,  -9.42857143,
        -7.71428571,  -6.        ,  -4.28571429,  -2.57142857,
        -0.85714286,   0.85714286,   2.57142857,   4.28571429,
         6.        ,   7.71428571,   9.42857143,  11.14285714,
        12.85714286,  14.57142857,  16.28571429,  18.        ,
        19.71428571,  21.42857143,  23.14285714,  24.85714286,
        26.57142857,  28.28571429,  30.        ,  31.71428571,
```

(continues on next page)

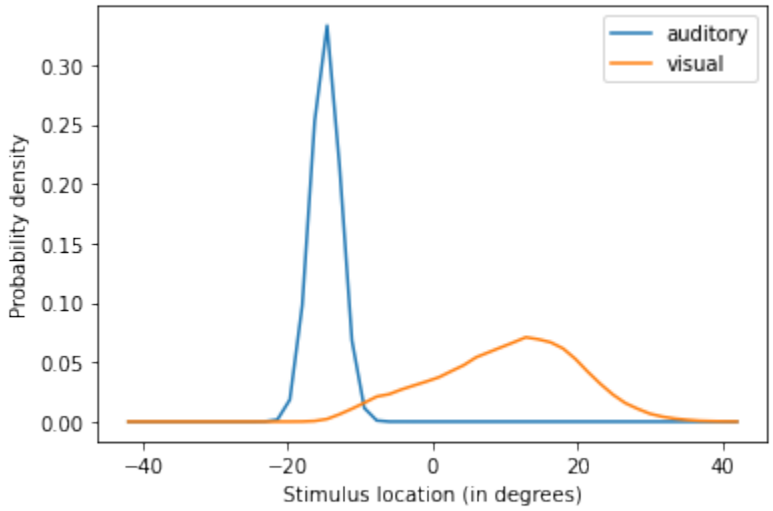
(continued from previous page)

```
       33.42857143,  35.14285714,  36.85714286,  38.57142857,
       40.28571429,  42.        ]), 1.7142857142857142), N=100000, auditory_sigma=2,␣
↪auditory_var=4, visual_sigma=10, visual_var=100)
```

By calling the model you can observe its main built-in parameters. You can refer to the documentation for more deatils about the available parameters.

Let's run the model for two conflicting stimulus locations:

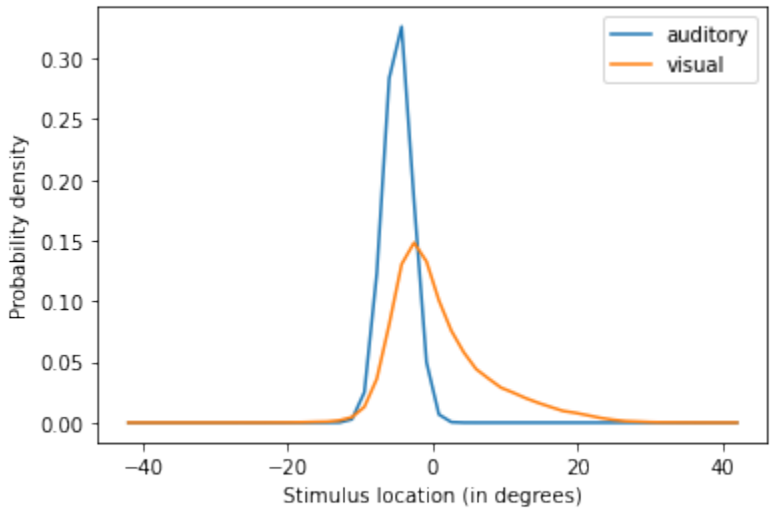
[21]:
```
res = model.run(auditory_location=-15, visual_location=15)

for k in out:
    plt.plot(model.possible_locations[0], res[k], label=k)
plt.legend()
plt.xlabel("Stimulus location (in degrees)")
plt.ylabel("Probability density")
plt.show()
```



The model outputs the unisensory estimates as being apart and centered around the given locations (-15 and 15 for the auditory and visual modality respectively).

Now let's see what happens if we reduce the distance of the stimuli:

[19]:
```
res = model.run(auditory_location=-5, visual_location=5)

for k in out:
    plt.plot(model.possible_locations[0], res[k], label=k)
plt.legend()
plt.xlabel("Stimulus location (in degrees)")
plt.ylabel("Probability density")
plt.show()
```



The model shows how the less precise stimulus location (in this case, visual) is biased towards the more precise one. This happens because the model is computing the unisensory estimates based on the probability of the stimuli originating from a common source.

> This demonstration of the Bayesian Causal Inference model mechanics is inspired in the tutorial of the BCIT Toolbox.

2.2.5 Build your own scikit-neuromsi model!

You can implement your own model by importing the `core` module and creating a class with the decorator `neural_msi_model`. Such decorated class must have two attributes: `stimuli` and `integration` as shown below:

> `stimuli` must be a list of functions (each representing an unimodal sensory estimator) and `integration` a function (representing the multisensory estimator).

[22]:
```
from skneuromsi import core


def unisensory_estimator_a():
    return "zaraza"


def unisensory_estimator_b():
    return "zaraza"


def multisensory_estimator():
    return "zaraza"


@core.neural_msi_model
class MyModel:

    # estimators
    stimuli = [unisensory_estimator_a, unisensory_estimator_b]
    integration = multisensory_estimator
```

You can provide further specifications to the model by including `hyperparameters` and `internals` in the model class:

[55]:
```
def unisensory_estimator_a(a_weight, baseline):
    u_estimate_a = baseline + a_weight * 2
    return u_estimate_a


def unisensory_estimator_b(b_weight, baseline):
    u_estimate_b = baseline + b_weight * 2
    return u_estimate_b


def multisensory_estimator(
    unisensory_estimator_a, unisensory_estimator_b, a_weight, b_weight
):
    return (
        unisensory_estimator_a * a_weight + unisensory_estimator_b * b_weight
    )


@core.neural_msi_model
class MyModel:

    # hyperparameters
    baseline = core.hparameter(default=100)

    # internals
    a_weight = core.internal(default=0.7)
    b_weight = core.internal(default=0.3)

    # estimators
    stimuli = [unisensory_estimator_a, unisensory_estimator_b]
    integration = multisensory_estimator


model = MyModel()
model.run()
```

[55]:
```
101.16
```

You can also configure parameters that are specific for each model run. For this purpose you can include them as parameters of the unisensory estimators functions (here as `input`):

[54]:
```
def unisensory_estimator_a(a_weight, baseline, input):
    u_estimate_a = baseline + a_weight * 2 + input
    return u_estimate_a


def unisensory_estimator_b(b_weight, baseline, input):
    u_estimate_b = baseline + b_weight * 2 + input
    return u_estimate_b
```

(continues on next page)

(continued from previous page)

```
def multisensory_estimator(
    unisensory_estimator_a, unisensory_estimator_b, a_weight, b_weight
):
    return (
        unisensory_estimator_a * a_weight + unisensory_estimator_b * b_weight
    )


@core.neural_msi_model
class MyModel:

    # hyperparameters
    baseline = core.hparameter(default=100)

    # internals
    a_weight = core.internal(default=0.7)
    b_weight = core.internal(default=0.3)

    # estimators
    stimuli = [unisensory_estimator_a, unisensory_estimator_b]
    integration = multisensory_estimator


model = MyModel()
model.run(input=5)
```

[54]: 106.16

For more details about model building, please refer to the documentation.

2.3 License

TRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

2.4 skneuromsi API

Scikit-neuromsi currently has three classes which implement neurocomputational models of multisensory integration.

The available modules are:

- **alais_burr2004: implements the near-optimal bimodal integration** employed by Alais and Burr (2004) to reproduce the Ventriloquist Effect.
- **ernst_banks2002: implements the visual-haptic maximum-likelihood** integrator employed by Ernst and Banks (2002) to reproduce the visual-haptic task.
- **kording2007: implements the Bayesian Causal Inference model for** Multisensory Perception employed by Kording et al. (2007) to reproduce the Ventriloquist Effect.

In addition, there is a **core** module with features to facilitate the implementation of new models of multisensory integration.

2.4.1 Module skneuromsi.alais_burr2004

2.4.2 Module skneuromsi.ernst_banks2002

2.4.3 Module skneuromsi.kording2007

2.4.4 Module skneuromsi.core

Implementation of multisensory integration neurocomputational models in Python.

class skneuromsi.core.**Stimulus**(*name*, *hyper_parameters*, *internal_values*, *run_inputs*, *function*)

Bases: object

Class for computing unisensory estimators.

name

Name of the estimator.

Type attr.ib

hyper_parameters

Set of hyperparameters coming from a neural_msi_model.

Type attr.ib

internal_values

Set of internal values coming from a neural_msi_model.

Type attr.ib

run_inputs

Set of inputs coming from a neural_msi_model run.

Type attr.ib

function

Callable that defines the computation of the unisensory estimate.

Type attr.ib

class skneuromsi.core.**Integration**(*name*, *hyper_parameters*, *internal_values*, *stimuli_results*, *function*)

Bases: object

Class for computing the multisensory estimator.

name

Name of the estimator.

Type attr.ib

hyper_parameters

Set of hyperparameters coming from a neural_msi_model.

Type attr.ib

internal_values

Set of internal values coming from a neural_msi_model.

Type attr.ib

stimuli_results

Set of inputs coming from unisensory estimators.

Type attr.ib

function

Callable that defines the computation of the multisensory estimate.

Type attr.ib

class skneuromsi.core.**Config**(*stimuli*, *integration*)

Bases: object

Class for configuring a neural_msi_model.

stimuli

List of skneuromsi.Stimulus that define the unisensory estimators of the neural_msi_model.

Type attr.ib

integration

A skneuromsi.Integration that defines the multisensory estimator of the neural_msi_model.

Type attr.ib

run_inputs

Set of inputs coming from a neural_msi_model run.

Type attr.ib

get_model_values(*model*)

Gets the hyperparameters and internals of the neural_msi_model.

run(*model*, *inputs*)

Executes the multisensory integration.

skneuromsi.core.**hparameter**(***kwargs*)

Creates an hyperparameter attribute.

Parameters ****kwargs** (dict, optional) – Extra arguments for the hyperparameter setup.

Returns Hyperparameter attribute.

Return type attr.ib

skneuromsi.core.**internal**(***kwargs*)

Creates an internal attribute.

Parameters ****kwargs** (dict, optional) – Extra arguments for the internal setup.

Returns Internal attribute.

Return type attr.ib

skneuromsi.core.**get_class_fields**(*cls*)

Gets the fields of a class.

Parameters **cls** (class) – Class object relevant for the model, usually built on top of an estimator.

Returns

- **hparams** (set) – Set containing the class attributes labeled as hyperparameters.
- **internals** (set) – Set containing the class attributes labeled as internals.

skneuromsi.core.**get_parameters**(*name*, *func*, *hyper_parameters*, *internal_values*)

Classifies the parameters of a function in hyperameters, internals or run inputs.

Parameters

- **name** (str) – Name of the function.
- **func** (callable) – Function to extract parameters from, usually an estimator.
- **hyper_parameters** (set) – Set containing attributes labeled as hyperparameters.
- **internal_values** (set) – Set containing the attributes labeled as internals.

Returns

- **shparams** (set) – Set containing the function parameters classified as hyperparameters.
- **sinternals** (set) – Set containing the function parameters classified as internals.
- **sinputs** (set) – Set containing the function parameters classified as run inputs.

skneuromsi.core.**change_run_signature**(*run*, *run_inputs*)

Modifies the signature of the run method of a neural_msi_model.

Parameters

- **run** (callable) – Function that delegates all the parameters to the run method of a skneuromsi.Config class.
- **run_inputs** (set) – Set containing the class attributes labeled as run inputs.

Returns **run** – Run method with a new signature including the run_input parameters.

Return type callable

skneuromsi.core.**neural_msi_model**(*cls*)

Defines a class as a neural_msi_model.

Parameters **cls** (class) – Class object of the model.

Returns **acls** – Class with a neural_msi_model setup.

Return type attr.s

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX