Generalized implementation of the Same Analysis Approach framework

Daniel Vargas

Supervised by: Kai Görgen

14.12.2018

BCCN

Contents

[Generalized implementation of the Same Analysis Approach framework 2](#_Toc532562986)

[Requirements 2](#_Toc532562987)

[Hardware requirements 2](#_Toc532562988)

[Operating System 2](#_Toc532562989)

[Software requirements 2](#_Toc532562990)

[Useful links 3](#_Toc532562991)

[Getting started 4](#_Toc532562992)

[Usage 4](#_Toc532562993)

[Adapting the code 5](#_Toc532562994)

[Architecture 7](#_Toc532562995)

[User files 9](#_Toc532562996)

[Configuration File 9](#_Toc532562997)

[Data 10](#_Toc532562998)

[Decoding sets file 11](#_Toc532562999)

[Description 11](#_Toc532563000)

[Expected values 12](#_Toc532563001)

[Output files 13](#_Toc532563002)

[API Reference 14](#_Toc532563003)

[SAA Interface 14](#_Toc532563004)

[Visualization 15](#_Toc532563005)

[MATLAB Add-on 15](#_Toc532563006)

[Workflow 17](#_Toc532563007)

[Sequence 17](#_Toc532563008)

[Activities 17](#_Toc532563009)

# Generalized implementation of the Same Analysis Approach framework

The use of novel statistical analysis methods, often employing complex data analysis pipelines (e.g. in the area of neuroimaging), has raised the question how to validate if a given set of experimental design and statistical analysis pipeline allows the expected statistical inference. The Same Analysis Approach (SAA; Görgen et. al 2018 [[neuroimage](https://www.sciencedirect.com/science/article/pii/S1053811917311072)], [[arxiv](https://arxiv.org/abs/1703.06670)]) is a framework that tests experimental variables and simulated data preserving the properties of the data analysis in order to achieve this.

This document describes a first generalized implementation of the approach. Because experiments can be very diverse, specific effort was put on a modular setup to allow users to easy adapt and expand the system to their needs.

## Requirements

In this section we postulate the software specifications where this package was tested. It is possible that the software can be executed in other environments that have not been explored yet, but the results have not been documented so far.

### Hardware requirements

* 32- or 64-bit computer.
* 2 MB disk space.

### Operating System

* Windows 7 or superior.
* Linux (Debian distributions supported).

In principle the software is cross platforms and should not have a problem running under any type of operating system, including macOS, and other Linux distributions. Further developers could enhance this project by extending the support to these operating systems.

### Software requirements

Full version

* MATLAB R2016b.
* MATLAB API for Python.
* Python 3.5, or 3.6.
* Google Chrome or Mozilla Firefox.
* MATLAB dependencies:
  + The Decoding Toolbox (TDT, Hebart, Görgen, etal, 2015 [<bccn-berlin.de/tdt>])
  + SAA Add-on
* Python dependencies:
  + Bokeh 0.13.0.
  + NumPy 1.15.2.
  + Pandas 0.23.4
  + SciPy 1.1.0.

PySAA (Python SAA framework)

* Python 3.5, or 3.6.
* Google Chrome or Mozilla Firefox.
* Python dependencies:
  + Bokeh 0.13.0.
  + NumPy 1.15.2.
  + Pandas 0.23.4
  + SciPy 1.1.0.

It is recommended to install and use these libraries within a virtual environment, like an Anaconda environment.

### Useful links

|  |  |
| --- | --- |
| Useful links for required software | |
| Anaconda | <https://conda.io/docs/user-guide/install/index.html> |
| Bokeh | <https://bokeh.pydata.org/en/latest/docs/installation.html> |
| MATLAB API for Python | <https://de.mathworks.com/help/matlab/matlab-engine-for-python.html> |
| TDT | <https://sites.google.com/site/tdtdecodingtoolbox/> |

Table 1. Some links are provided for most of the external software dependencies.

## Getting started

### Usage

After installing the software requirements, the user should have all the necessary User files. Then a terminal should be opened on the directory of the SAA Python Integration. The usage for the interface is:

> python interface.py [-h] [--no\_plot] cfg\_file [--no\_matlab] [--n\_subjects] [--n\_sessions]

positional arguments:

cfg\_file JSON file with the configuration information

optional arguments:

-h, --help show this help message and exit

--no\_plot Do not produce HTML output figure

--no\_matlab Do not use MATLAB, use pySAA version.

--n\_subjects Specify number of subjects (valid for pySAA).

--n\_sessions Specify number of sessions (valid for pySAA).

#### Quick tutorial: pySAA.

The only external files needed in here are those corresponding to the BIDS project folder. In this case we should have a known number of subjects that have a fixed number of sessions. As this implementation is a basic setup, it does not include all the necessary features, such as scanning for the files through the project folder. For this reason, the project folder should be given as a string that will be formatted with the arguments number of subjects and number of sessions. According to the specification the folder BIDS\_pySAA should have the structure: BIDS\_pySAA/sourcedata/sub-X/sess-Y/fmri/data.tsv. For example, if there are two subjects with 4 sessions each, the program should be run:

> cd path-to-pySAA

> python interface.py ~/Documents/BIDS\_pySAA/ --no\_matlab --2 --4

Which will produce the output figure:

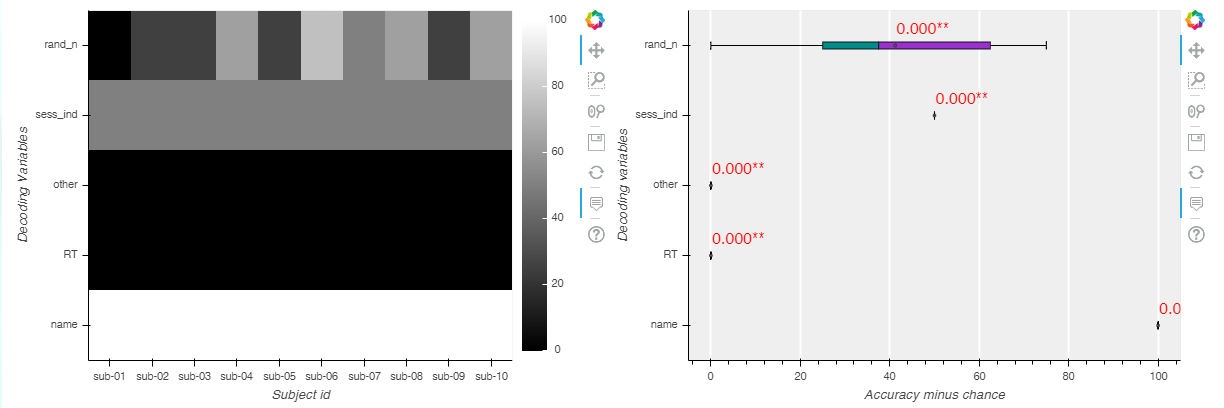


Figure 1. Output of the simple example produced by the pySAA module.

#### Quick tutorial: Full version

We start with a folder structure illustrated as it follows:

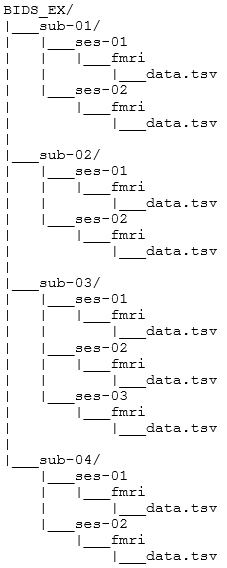
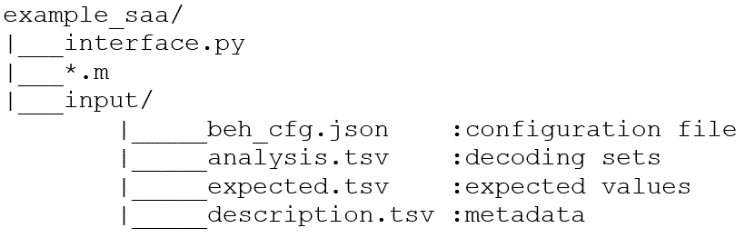


Figure 2. Example of folder architecture. Left side shows the folder where the MATLAB and Python scripts are, as well as a folder to place input files. On the right side an example of a BIDS folder is shown.

On the left side we see the folder structure where the Python Integration for SAA is, and on the right side we see an example of a BIDS folder architecture.

When we open a terminal on the folder example\_saa, and run:

> python interface.py ./input/beh\_cfg.json

The terminal will indicate when every sub process has started and finished, plus we will be able to see the output of the MATLAB commands. If every step was executed correctly, the default web browser will display the html file with the output figure.

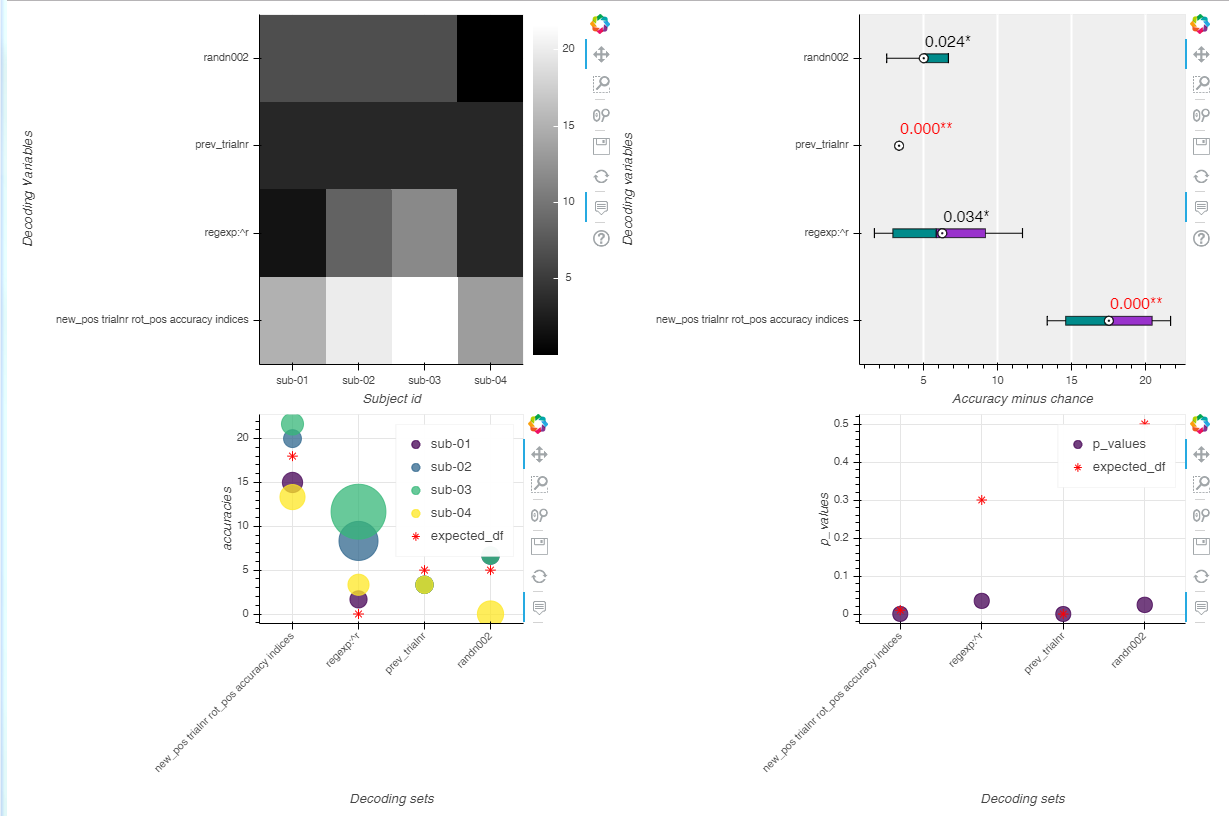


Figure 3. Example of an output image created when running the decoding analysis on the provided example.

#### Quick tutorial: New decoding

To process the data in a different way than the one showed by the pySAA example one could modify the process subject function in pySAA.py:

process\_subject**(**general\_file**,** sub\_ind**,** sessions**,** cv**,** plot\_cv**=**False**)**

Right now the decoding happens in these lines:

**for** field **in** result**:**

data **=** np**.**array**([**result**[**field**]]).**T

score **=** np**.**mean**(**cross\_val\_score**(**clf**,** data**,** labels**,** cv**=**cv**,** groups**=**groups**))**

scores**.**append**(**score**)**

One could choose instead a t-test value and the median of the runs to analyze the information. One could also have the case where the decoding sets should be the combination of all variables. In order to do that the reordering should be:

**for** product **in** itertools.product(result)**:**

data **=** np**.**array**([**result**[**product**]]).**T

score **=** np**.**median**(**scipy.stats.ttest\_ind**(**clf**,** data**,** labels**,** cv**=**cv**,** groups**=**groups**))**

scores**.**append**(**score**)**

#### Quick tutorial: New data

A new experimental data setup can be achieved following these steps:

1. Create a BIDS folder for the project.
2. For every session of every subject make an fmri folder with a data.tsv file. This file should look like this: (column names are variables, and every row is a trial)



1. In a separate folder, e.g. ~/input create a file called description.tsv where the first column of every row has SAA variables, the second column the value with which defaults are replaced, and the third column the type of the variable (ordinal, interval).



1. In the same folder create a file called analyses.tsv, where every row is a decoding set, and every value is an SAA variable



1. Lastly create a file called beh\_cfg.json with this content:

{

"path": {

"value": "\BIDS\_ex\sourcedata",

"type": "other"

},

"cfg\_mat\_file": {

"value": "\tmp\beh\_cfg.mat",

"type": "other"

},

"decoding\_sets\_file": {

"value": "~\input\analyses.tsv",

"type": "other"

},

"description\_file": {

"value": "~\input\description.tsv",

"type": "other"

},

"saa\_path": {

"value": "\toolboxes\SAA\_rev\_aeffect",

"type": "other"

},

"tdt\_path": {

"value": "\toolboxes\decoding\_toolbox\_v3.994",

"type": "other"

},

"output\_data": {

"value": "\tmp\data\_mat.mat",

"type": "other"

},

"output\_result": {

"value": "\tmp\result.mat",

"type": "other"

},

"use\_own\_decoding": {

"value": "false",

"type": "object"

},

"previous\_on": {

"value": "true",

"type": "object"

},

"labelnames": {

"value": "{'1' '2'}",

"type": "object"

},

"unbalanced\_data": {

"value": "ok",

"type": "other"

}

}

### Adapting the code

In some cases, the user might have the need of modifying or adding functionalities to: perform the decoding in a different way than the one provided, adding plots to the expected output figures, and including new types of visualization. We present here some guidelines on how to do this, however it is highly recommended that the whole technical specification is read and understood before altering the existing code.

#### Different decoding

The use of a different decoding function is currently supported. To ensure a proper integration, there should be paid special attention to the input and output parameters. For the input at least three parameters are passed:

* Decoding set: which is a cell array where each element is the name an SAA variable.
* Data: structure with the data of a particular subject. The structure can be observed in the Format remarks section.
* Cfg structure: which carries the necessary parameters for decoding.

The output should consist of:

* Result: This is a structure whose minimal requirements is to have a nested structure called accuracy\_minus\_chance where the field output has the decoding information for that particular set of SAA variables.
* Decoding cfg: which is the cfg structure with the modifications that were performed on the decoding script if any.

This function should be saved on the same working folder, or in a subfolder of it, and specify it on the configuration file. See User File Content for more information. To have a more detailed idea of the implementation, the script process\_subject.m is a good place to start.

#### Expected values

The current version can produce two figures that show how the actual data differs from an expected value. This two figures correspond to the accuracies calculated in the decoding function, and to the p-values calculated by performing a second level analysis on the same accuracies. If further measures are wanted, then the TSV file should have a column with this measure, and the actual data should be calculated and included in the results variable of the Visualization.visualize module as a pandas DataFrame. For example, if the mean values would like to be observed, one should include a column of expected mean values on the TSV file, with a header called mean; then the results dictionary should also have a line like this one: mean **=** pd**.**DataFrame**(**mean**,** index**=**kwargs**[**"sets"**],** columns**=[**"mean"**])**.

#### Adding plots to the output figure

If the user would like to include more plots to the output figure, he should only bear in mind that the plots must be either bokeh objects or a fixed image. In either case he should incorporate his plot inside a column layout like the existing ones.

For example: show**(**column**(**existing\_fig1, existing\_fig2**,** new\_fig1**)).**

## Architecture

This software has a modular architecture whose main components are a python interface which acts as a blackboard, maintaining a structured global representation of the solution objects. These objects are passed by the blackboard to different modules, MATLAB for preprocessing and decoding, and python modules for visualization.

The MATLAB code is built on the top of the previous alpha SAA software package in MATLAB and it also makes use of *The Decoding Toolbox* (TDT; Hebart, Görgen etal 2015 [<bccn-berlin.de/tdt>]).

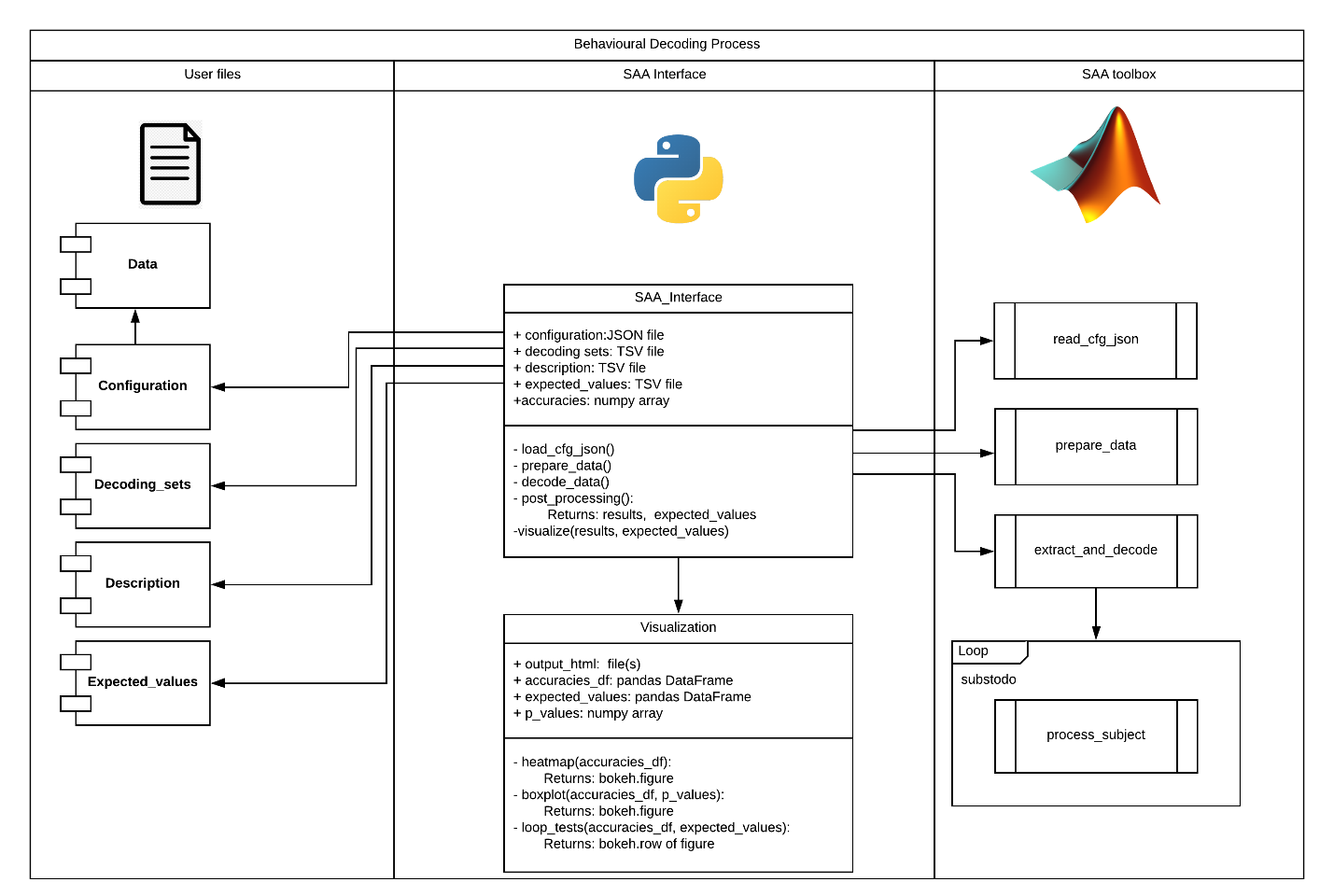


Figure 4. Software Architecture for the SAA (formerly: Behavioural Decoding) Process.

The current version has two python classes, which correspond to the SAA\_Interface and Visualization classes of the diagram shown in *Figure 4. Software Architecture for the SAA (formerly: Behavioural Decoding) Process*. The MATLAB code is a set of scripts on a folder, and it works with the existing SAA add-on. For further work in this project, an object oriented paradigm is recommended, in order to have a more maintainable and scalable code. On the class diagram of *Figure 5. Class diagram of the suggested object oriented design*. a design proposal is shown. This design would keep the current business logic, only changing the architecture of the MATLAB package.

NOTE: There are two things to be noted for this implementation. First, one should be aware that instances of classes cannot be interpreted by the Python API, one should rather instantiate classes on the MATLAB engine, and make this variables part of the existing workspace. Second, while doing this implementation it might be a good time to consider changing the communication between modules, which currently is done through files, and it could be optimized by using the Python API.

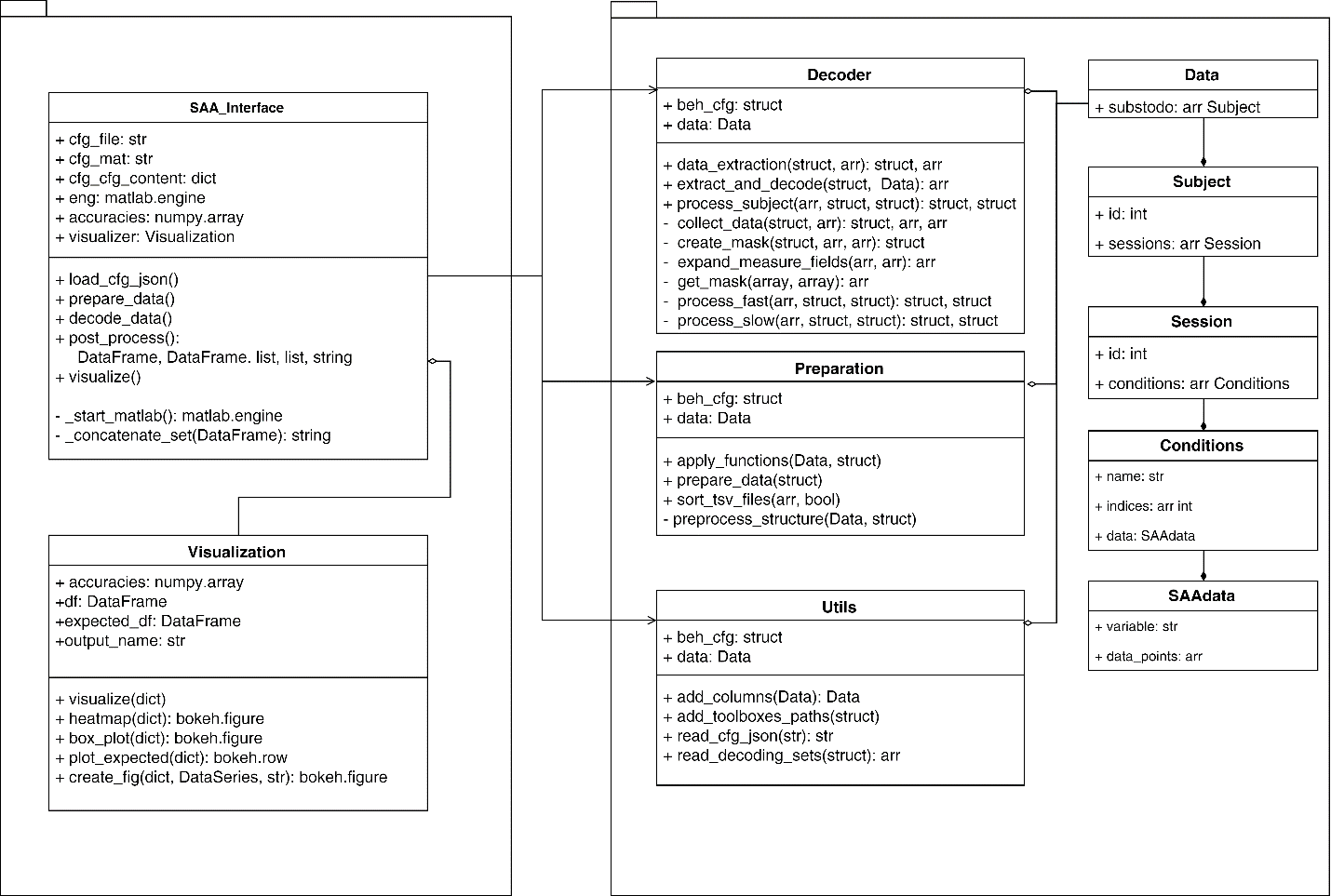


Figure 5. Class diagram of the suggested object oriented design.

## User files

### Configuration File

This is a JSON file that contains the necessary parameters for the SAA process. This is the only file that will have to be explicitly specified by the user to the interface.

#### Format

The content of the file must have the following format:

{

key1: {

"value": value1,

"type": type1

},

key2: {

"value": value2,

"type": type2,

subfield: sub\_value

},

}

It is a collection of keys whose values are objects with two mandatory fields: *value* and *type*. Additional subfields can be included if needed. This configuration file will be converted into a mat file when the interface calls the load\_cfg\_json function. This conversion is essentially transforming the JSON file into a MATLAB structure, where the fields are the names of the JSON keys, and the values are computed in the following way.

* For "type": "object" the value will be the MATLAB evaluation of the value of the key, for example: field: {"value": value,"type": object} this will be: struct.field = eval(value).
* The case where "type": "function" is a special one. The structure has a field called functions which is an array of structures with a name, a function handler, and additional arguments. The purpose of this functions is to apply sequentially a series of functions to the data, e.g. add a column of ones, or n columns of random values. In this case a third field named "args" must be included, and its content has the format: "{@foo, arg1, ..., argN}"}.

The first argument is a handler that references an existing MATLAB function, either built-in or provided by the user. An example of this case is the following:

field: {"value": name, "type": function, "args": "{@foo, arg1, arg2}"} which assumes that the function foo exists, and this will yield to the structure to be:

struct.functions.func = name

struct.functions.args = eval([@foo, arg1, arg2])

* Any other type would imply a simple assignment:

field: {"value": value, "type": other} will be: struct.field = value.

#### Content

There are some subgroups in which we could classify the different fields of this JSON file: paths to input files and toolboxes, paths to output files, functions to apply to data, and parameters useful for the decoding process.

|  |  |  |
| --- | --- | --- |
| Category | Field name | Description |
| Inputs | path | Path to BIDS data folder where the data is |
| decoding\_sets\_file | Path to TSV file that contains the sets of decoding variables. |
| description\_file | Path to TSV file of metadata. |
| expected\_values | Path to TSV file of expected values. (optional) |
| Outputs | output\_data | Path where the data will be saved as a mat file. |
| output\_result | Path where the results will be saved as a mat, tsv, and html files. |
| cfg\_mat\_file | Path where the configuration file will be stored as a mat file. |
| Toolboxes | saa\_path | Path to SAA MATLAB software add-on. |
| tdt\_path | Path to TDT MATLAB software toolbox. |
| Functions | \*include as many as desired | Functions to be applied to the data. |
| Parameters | use\_own\_decoding | If the user wants to use his own decoding function, this should be ‘true’ and the field decoding\_function should be the name of such script. (optional) |
| decoding\_function | Name of the script to use as a decoding function. (optional) |
| previous\_on | This value should be ‘true’ if one wants to include the analysis of the previous trial. (optional) |
| unbalanced\_data | If data is intentionally unbalanced this key should have a value of: ‘ok’. (optional) |
| substodo | List of subjects to process. |
| labelnames | Names of the conditions or values that the decoding will use, formatted as a cell array. E.g. “{‘a’, ‘b’, ‘c’}” |

Table 2. Content of the configuration file grouped by category.

Notes: The current version uses output files to persist and communicate data across the blackboard interface and MATLAB processes. For further developments, it should be considered to avoid this step by passing this structures as arguments through the Python API, minding the fact that the support for function handlers, nested structures, and multi-dimensional arrays is little or non-existent.

### Data

The data used for the decoding process using the SAA, should indeed be processed just as if it were fMRI data itself. One standard used to store neuroimaging data is called BIDS. This standard specifies how the folder structure where the data is should be structured, and which files should be present. Following this standard, the experimental data should also be stored after this specifications, in the form of TSV files, one for every session of every subject. To know more about the specification please refer to: <http://bids.neuroimaging.io/>.

Important remarks about using TSV files: each subject’s session will have a file (two dimensional array) where each row corresponds to one trial, and the trials are ordered in a consecutive fashion. The columns of the file represent experimental variables which could be potential sources of confounds, they are also named SAA variables. If one data point is unknown or unavailable it is important to specify this as n/a, instead of leaving this blank. On further steps this could be replaced for a determined value.

#### Format remarks

The data files should be under the folder:

sub-N/ses-M/fmri/data.tsv if there are multiple sessions or:

sub-N/fmri/data.tsv if there is only one session.

NOTE: further developments might work on the flexibility for different filenames, but for the current version, only those files that follow this structure will be taken in account.

Internally the data is stored by MATLAB in a struct object, which has the following architecture:

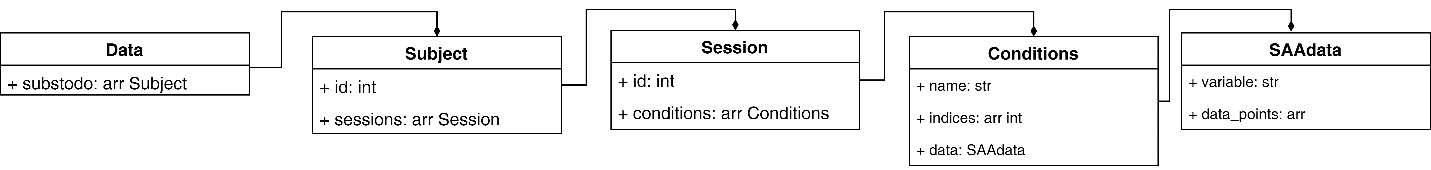


Figure 6. Architecture of the MATLAB structure used to store the data.

### Decoding sets file

The decoding process of a set of variables gives as an output value the accuracy minus chance level. However, we usually want to explore more than one variable or sets of variables. In order to specify which are these sets, we create this TSV file where each row represents one decoding set. The values of each row are SAA variables, which should exist in the columns of the input data.

### Description

This TSV file contains metadata necessary to know how non existing values should be handled, and which variables are ordinals. Generally speaking, experimental variables can be classified in three classes: categorical, ordinal, or interval. Categorical variables have multiple categories but there is no intrinsic ordering to them. Ordinals can have some intrinsic ordering, but the spacing might be uneven, or hard to quantify. Interval variables have a clear numeric ordering. Since sometimes categorical and ordinal variables can be exchanged, depending on how the modeler specifies them, we will assume only two classes: intervals for numerical data with intrinsic ordering, and ordinal variables that lack this order.

The format for this file is separated by tabs (TSV) where each row corresponds to each one of the used decoding variables. In every row, three values should appear: variable name, default value, data class.

The defaults will replace the values of the variable marked as n/a during function prepare\_data. If the data class of the variable is ordinal it will be expanded then.

NOTE: The current version does not expand these variables. This is one of the most urgent points to develop for further releases.

### Expected values

An optional file that has the information of the expected outcome can be placed as a TSV file, where each row corresponds to every decoding set. The first column should be the decoding variables of the set, separated with spaces (NOTE: it is actually important that this series of variables is correctly written as the first value of the row in order to plot correctly; further developments could enhance this function in order to make it more robust).

A header is also required to specify the names of the outcome variable to be tested, e.g. p\_values, accuracies.

|  |  |  |
| --- | --- | --- |
|  | accuracies | p\_values |
| trialnr accuracy pos | **18** | **0.01** |
| regexp:^r | **0** | **0.3** |
| prev\_trialnr | **5** | **0** |
| randn002 | **5** | **0.5** |

Table 3. Example of the TSV file with expected values.

## Output files

The current version of this software produces a series of output files which are necessary for the communication between the blackboard and the MATLAB modules. The reason behind this was the limited support of MATLAB data types in the Python API. As this API evolves, and other developers get involved in this project, they are invited to provide alternatives to this solution. On the other hand, persisting data is a good idea in this case since in this way some computations might be avoided.

Currently the files created by the SAA software are:

* cfg\_mat\_file: Mat file with the content of the parsed configuration file
* output\_data: Mat file with a structure that contains the totality of the data grouped by the variable named name, which usually corresponds to the experimental condition.
* output\_result.mat: Mat file with the structure with all the information obtained during the decoding process
* output\_result.tsv: TSV file with the decoding results. Every row corresponds to a decoding set, where the first value is the list of decoding variables, and the following values are the accuracies minus chance level, each column representing one subject.
* output\_result.html: HTML file with the output figure. It is created unless it is specified not to (via script parameters command line).

## API Reference

### SAA Interface

This is class serves as an orchestrator as it provides the necessary methods to read input files, prepare the data, perform the decoding analysis, and visualize the results. It keeps in memory the necessary variables to coordinate the information flow, and commands submodules to persist information to enhance communication.

SAAInterfaceWhen initialized, the module argparse handles the command line arguments and starts the MATLAB Engine API for Python.

The command line arguments of the current version are:

* cfg\_file (positional argument): path to the configuration file.
* --no\_plot (optional flag): if this flag is passed, no plot will be created

SAAInterface.\_start\_matlab**()**

This method is used when the class is instantiated. It creates a MATLAB process and returns a Python object to pass data and call functions executed by MATLAB. For further reading please refer to:

<https://www.mathworks.com/help/matlab/matlab_external/start-the-matlab-engine-for-python.html>

SAAInterface.load\_cfg\_json**()**

Using the MATLAB engine to call the function read\_cfg\_json, with the configuration file path as a parameter, it gets the path to a mat file from it. This file is loaded as a dictionary, and it has all the content of the original JSON file, only rearranged in a more convenient way.

See the section Configuration File for more information.

SAAInterface.prepare\_data**()**

The MATLAB engine calls the function prepare\_data, with the configuration file path as a parameter, to scan the BIDS folder to read the data and structure it in a more convenient format. The data is stored as a mat file according to the configuration file.

NOTE: the internal unction preprocess\_struct does not currently expand the ordinal variables. Further developments should implement that in this step, after the default values replacement.

SAAInterface.decode\_data**()**

The MATLAB engine calls the function decode\_data, with the configuration file path as a parameter, to perform the analysis process according to the specified decoding sets. The entire result is stored as a mat file, and it retrieves the output value to store it as a numpy array in the field self**.**accuracies.

SAAInterface.\_concatenate\_set**(**analysis**)**

Concatenate columns of non-null pandas Data Series called analysis.

SAAInterface.post\_process**()**

Using the dictionary that came from the configuration file it parses the TSV files with the decoding sets (to extract the sets of variables) and the one with the expected values if existent (and converts it into a pandas Data frame). It also stores the accuracy values as a TSV file.

It returns df**,** expected\_df**,** columns**,** sets**,** output\_html which correspond to: a data frame with the accuracies, a data frame with the expected values, a list with generated subject names, a list with strings corresponding to each decoding set, and the html file where the output figure will be displayed.

SAAInterface.visualize**(**df**,** expected\_df**,** output\_html**,** columns**,** sets**)**

It the flag –no\_plot was not set, it creates an instance of the Visualization package.

### Visualization

This package is a collection of useful functions that use the bokeh library to produce interactive plots. To know more about bokeh, please refer to: <https://bokeh.pydata.org/en/latest/>.

Visualization **(**accuracies**,** df**,** expected\_df**,** output\_name**,** **\*\***kwargs**)**

This is the main plotting class used by the SAA Interface. It calls the visualize function to create the main figures. The plots are saved under output\_name.

Visualization.visualize**(**kwargs**)**

It creates two plots first: a heat map of accuracies, and a box plot of a second level analysis of this accuracies. If the data frame Visualize.expected\_df exists, then it iterates over the columns of the data frame and creates a figure that compares the actual data with the expected one.

Visualization.heatmap**(**params**)**

Plots a 2D array of Visualization.accuracies as a heat map using bokeh. The dimensions of the numpy aray accuracies should be n x m, n=number of decoding sets, m=number of subjects. The dictionary params should have the keys sets and columns. It returns a bokeh image.

Visualization.box\_plot**(**params**)**

Returns a bokeh figure with a box plot created with bokeh with the second level analysis of Visualization.df. To perform such analysis the function scipy**.**stats.ttest\_ind was used. The dictionary params should have the key sets.

Visualization.create\_fig**(**data**,** expected\_series**,** test\_str**)**

The dictionary data should have real data points under the key test\_str. This points are plotted in a bokeh scatter plot, as well as the points in expected\_series. The difference between the data points and the expected\_series determines the size of the points in the graph.

Visualization.plot\_expected**(**results**)**

This function loops over the series of expected values, and calls create\_fig to create figures that compare expected values against real ones. It is returned in a bokeh row layout with one figure per data frame column. The dictionary results contains data frames of real data.

### MATLAB Add-on

The following specification corresponds to the scripts that are built on the top of the existing SAA software package.

add\_columns **(**data**,** func**,** colname**,** ncols**)**

This function is an example of a user defined function. It adds ncols of name colname to the data. The data points are created with function func.

add\_previous **(**data**)**

This function duplicates the existing columns with the data points shifted to the previous trial.

add\_toolboxes\_paths**(**beh\_cfg**)**

Adds path of SAA and TDT toolboxes and their subfolders.

extract\_and\_decode**(**beh\_cfg, data**)**

This function consists of a loop over the subject data, from which the data points that correspond to the decoding variables are extracted and decoded. It saves in a mat file the full information of the decoding process, and a float array with the output is returned.

data\_extraction**(**beh\_cfg, data**)**

This function gets the data points corresponding to the decoding sets for all the session of a given subject. It uses this information to create the data masks, and an array of regressor names, used for decoding.

prepare\_data**(**beh\_cfg**)**

This function parses data from tsv files into a data structure that can be used for decoding. It obtains parameters by reading from beh\_cfg which can be either a structure or the path to it. The data undergoes a preprocessing step in which non existing values are replaced by specified default values, and ordinal variables are expanded. Also, if the configuration file includes user specified functions, they are applied to the data. Finally, the data is saved on the specified output path.

Input: beh\_cfg: struct with parameters necessary for the data processing, or path to this struct.

process\_subject **(**decoding\_sets**,** Sess**,** cfg**,** beh\_cfg**)**

This is a wrapper function that chooses between two options for decoding. The faster option extracts data and masks from the decoding sets and performs a decoding procedure on the data. The slower option loops over each decoding set and applies a user specified decoding function to the data.

read\_cfg\_json**(**fname**)**

Script that reads a JSON file with the configuration information from the experiment pipeline, and places that information into a structure to be used in the course of the SAA process. It saves the output of this function on a mat file.

read\_decoding\_sets**(**beh\_cfg**)**

This script reads a TSV file specified in the beh\_cfg.decodings\_sets\_file that has the information of which variables will be used as decoding measures. The rows of this TSV should have in each cell the name of the variable to be used, or a regular expression that could match one or more variables.

sort\_tsv\_files**(**paths**,** previous\_on**)**

This function parses a set of TSV files into a structure of the form:

subject(subj\_ind).Sess(sess\_ind).U(u\_ind).SAAdata(data\_ind)

The variable paths is a cell array which rows are the paths to the sessions for a specific subject. previous\_on is a Boolean that indicates whether previous values of the variables would be included as new fields.

## Workflow

### Sequence

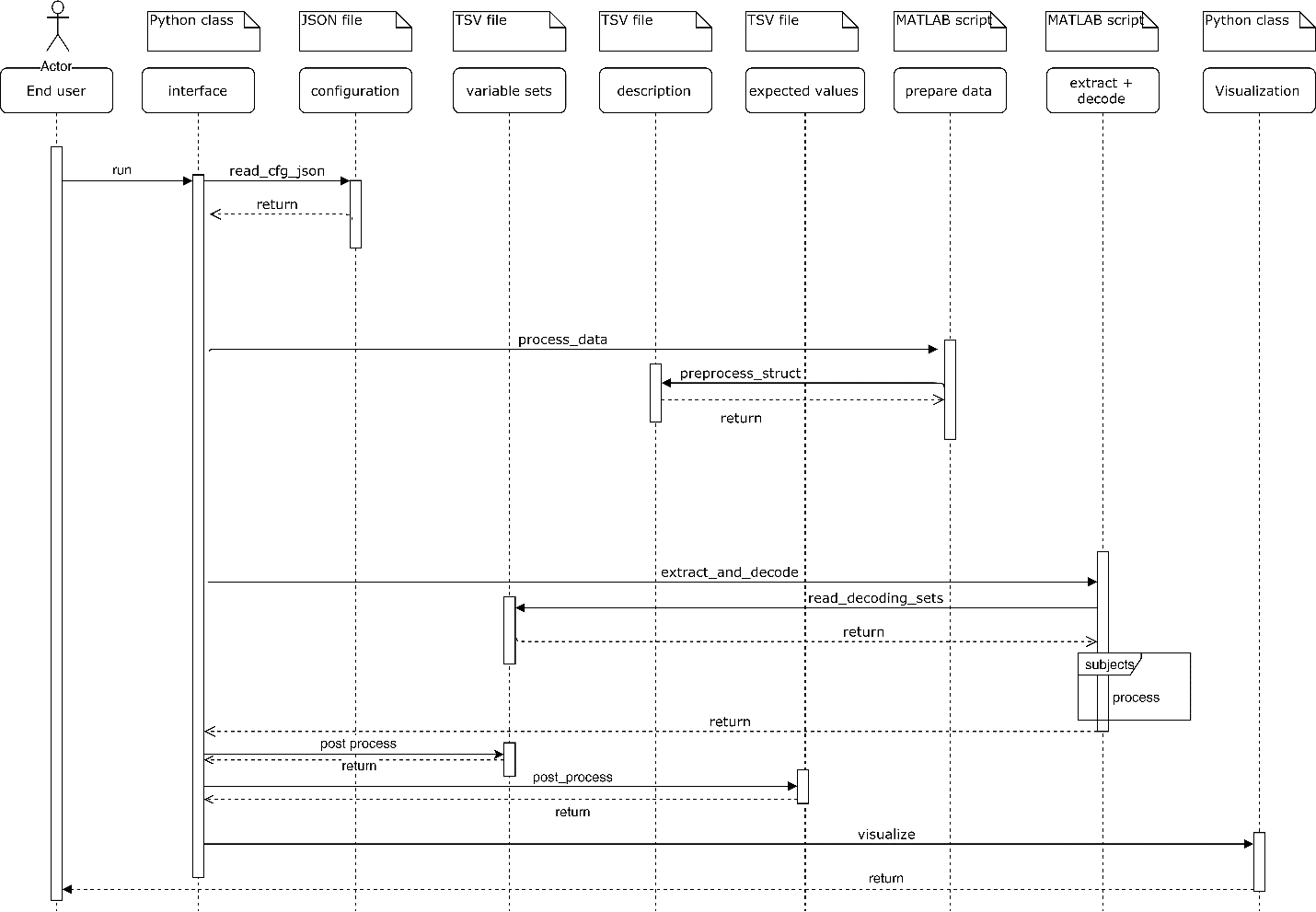


Figure 7. Sequence diagram that describes the basic workflow of the generalized implementation of SAA.

The diagram in Figure 7. Sequence diagram that describes the basic workflow of the generalized implementation of SAA. displays the sequence of activities that the different components realize. The user runs the program and the visualization output is presented to him, whereas the interface module acts as the orchestrator. Overall, five different processes can be distinguished, starting from the lecture of the configuration file, whose information is used by the interface. Then the preparation of the data takes place, followed by the decoding stage which loops a decoding process over the different subjects. By the end of this point the interface already has the actual output of the decoding, but it still calls a post processing routine to save the data and prepare it for visualization. Producing the output figure is the final step.

### Activities

The main activities of the different components are showed in Figure 8. Activity Diagram of the generalized implementation of SAA., and the sub processes are detailed in Figure 9. Detail activity diagram of the MATLAB sub processes.

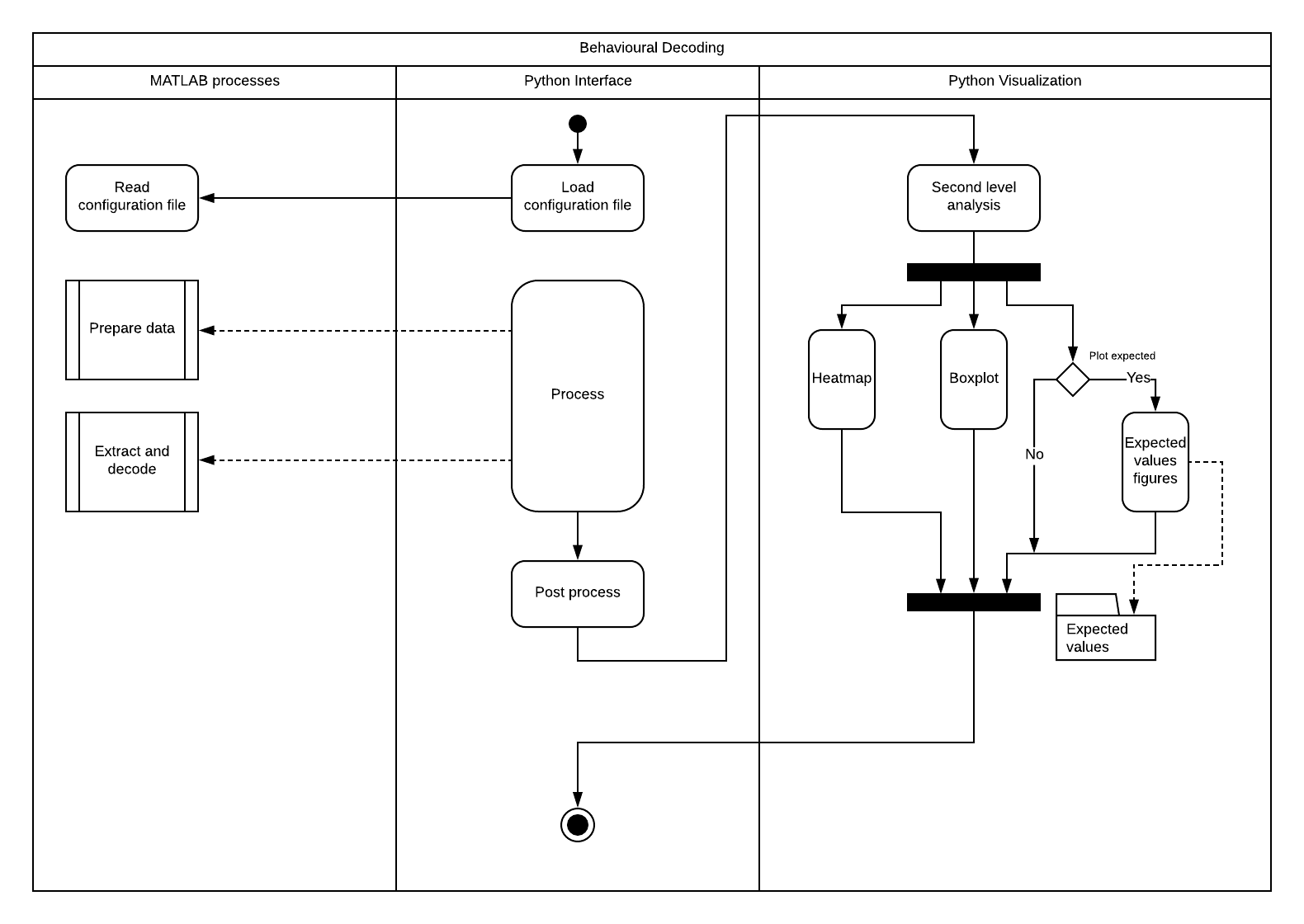


Figure 8. Activity Diagram of the generalized implementation of SAA.

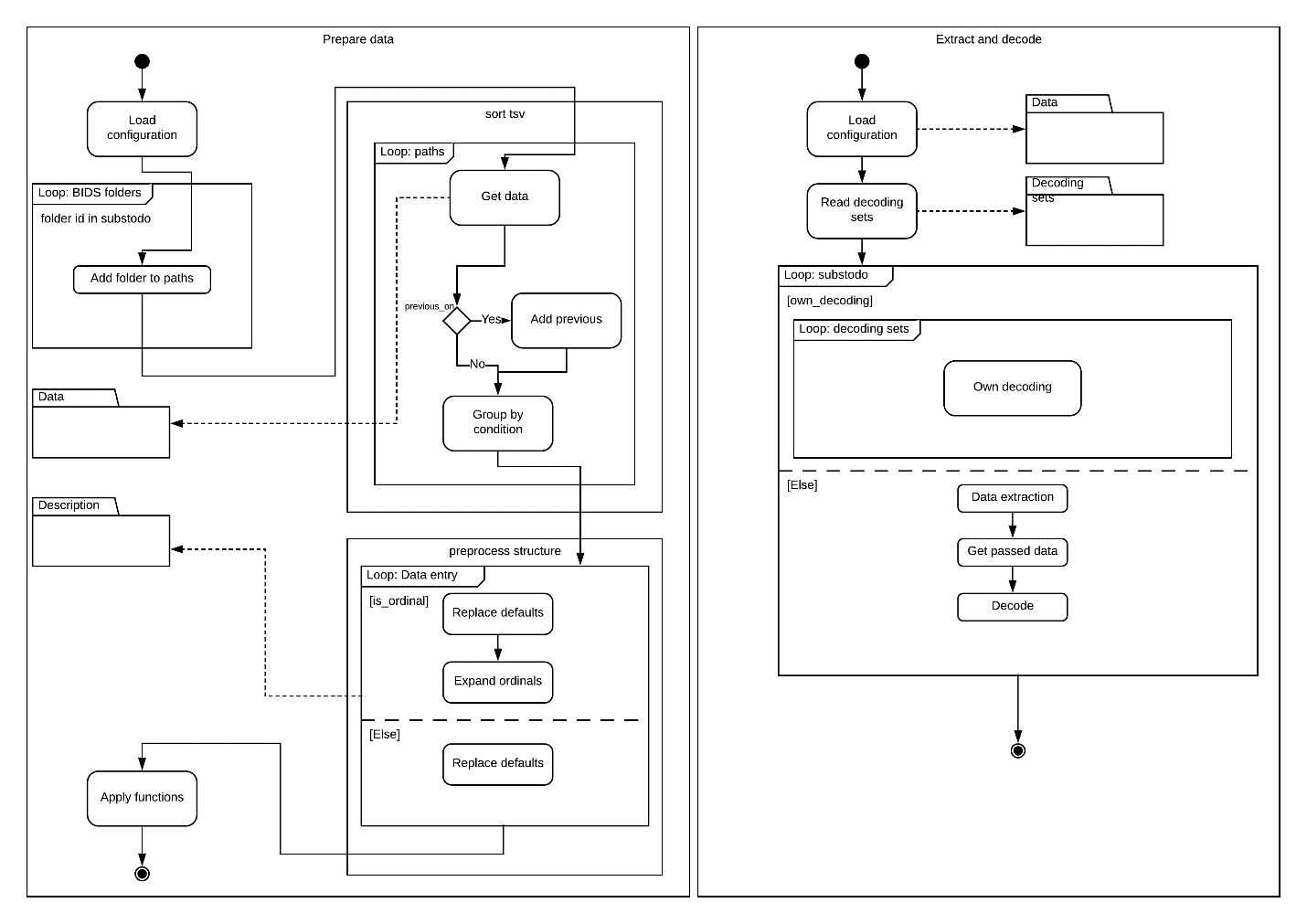


Figure 9. Detail activity diagram of the MATLAB sub processes.