Operation Manual

Lokaverkefni

T-404-LOKA

Reykjavik University - School of Computer Science,
Menntavegi 1, IS-101 Reykjavík, Iceland

Anton Björn Mayböck Helgason

Björgvin Ægir Elisson

Eva Björg Nåbye

Margrét Sól Aðalsteinsdóttir

Instructors: María Óskarsdóttir and Benedikt Hólm Þórðarson

Examiner: Anna Sigríður Íslind

May 13, 2022

Contents

1 Installation and Build

To install Arora, both Python and pip are required. Arora is installed by using the "pip" or "pip3" command, depending on the operating system in use. Here are examples of how to install the package depending on the operating system. The latest version of Python can be found here: `https://www.python.org/downloads/` Newer versions of Python 3 (3.8 and higher) come with pip installed already.

Unix

```
pip3 install arora
```

Listing 1: arora installation for unix

Linux and Windows

```
pip install arora
```

Listing 2: arora installation for linux

1.1 Building the Package

Commands to build the package but these depend on the operating system of the computer used to build them on. If build and wheel are already installed then skip those steps.

Windows

```
py -m pip install build
py -m pip install wheel
py -m pip build
py -m build --sdist
py -m build --wheel
```

Unix/Linux:

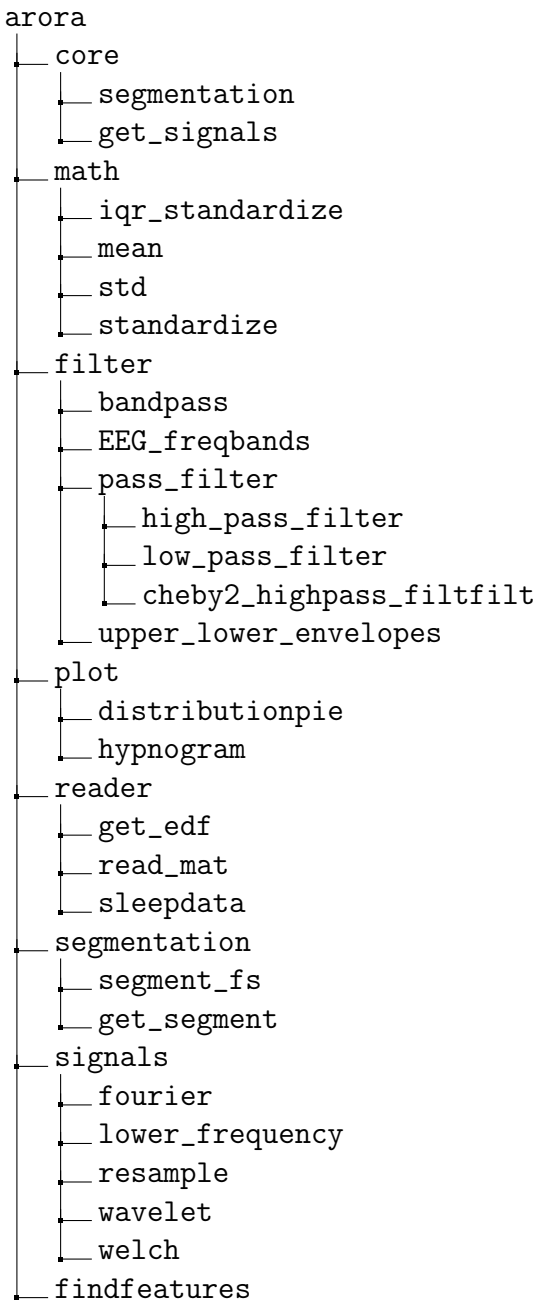
```
python3 -m pip install build
python3 -m install --wheel
python3 -m build
python3 -m build --sdist
python3 -m build --wheel
```

Both:

```
twine check dist/*
Username: __token__
Password: <the token value, including the $‘pypi-‘$ prefix>
```

2 Structure

Arora is split up into different submodules that each perform different functions when working with signals.



2.1 Core

The core includes all functions that will only be called once and the foundation functions that are to be used.

2.2 Filter

The filter includes pre-processing functionalities where signals are put through filters.

2.3 Signal

The signal includes pre-processing functionalities on the signal or some specific segment of the signal.

2.4 Math

Math includes the basic mathematical functionality of signals.

2.5 Plot

The plot includes all plotting and visualization functionality of the package.

2.6 Segmentation

Segmentation includes helper functions and stand-alone functions for segmentation. These functions are different from the segmentation function located in the core as they revolve around getting specific segments and not creating a signal with fixed segmentation.

2.7 Reader

The reader includes all functions and helper functions for reading, loading, and exporting data from different files and file types. Currently, there has only been implemented a function that reads EDF files but future functionality for mat files would go in this section.

2.8 Function Structure

The parameters should be ordered as follows:

1. File (or the path to it).
2. Signal list.
3. Sample frequency, default should be 250.0. If there are two or more frequencies, they should be in ascending order.
4. Cutoff (for filtering), lower cutoff first, then upper, when appropriate.
5. When segmenting, onset should come before duration.
6. Other parameters (depending on functions).
7. Filter order, when appropriate.

It is important when new functions are added, to use the typing Python module on the parameters. This is done to make the code more readable and to enforce typing rules not generally found in Python. Make sure the function has a return type for the same reasons the parameters should be typed. All parameters and function variables should have descriptive names.

3 Programming Rules

Arora is written in Python. The package follows the PEP8 style guide which is the official Python code convention and snake case naming convention is used. More information about the PEP8 style guide can be found here: `https://peps.python.org/pep-0008/`.

4 Documentation

The documentation has been published on Read the Docs, a public open-source website that enables automatic build, versioning, and hosting of software documentation. More information on Read the Docs can be found here `https://docs.readthedocs.io/en/stable/`.

4.1 Language

For the documentation, we went with reStructuredText which is an easy-to-read plaintext markup syntax and parser. It is useful when using in-line code documentation, such as Python docstrings as this was designed specifically with that in mind. Our documentation is written in reStructuredText within the `.../src/arora/source/docs/source` folder.

4.2 Updating Docs for Latest Release

Read the Docs enables automatic build and thus the updating of documentation can be done by pressing `build` on the home page of the project.

Figure 1: image of the build button on readthedocs.com

4.3 Structure

In terms of formatting, the documentation is nearly identical to Arora's source code; Where each subsection is determined by a submodule in the code. Each submodule has its own rst file which includes documentation on all of their respective functions. Outliers such as installation and other functionality that is not in its own submodule are split up into `installation` and `other`. This structure is subject to change depending on the complexity of the package in the future.

4.3.1 Standard

For each function, there is detailed documentation that includes a reference to the respective function. The documentation is structured as follows: Title of the function in full, the function with all its parameters highlighted, explanation of the parameters and returns of the function which include type and description, and then finally an example of the function in practice using dummy data.

- Title of the function
- The function and its parameters
- Description of the parameters
- What is returned
- Example of usage

To see all this in practice, refer to the excerpt below from Arora's documentation on the function `bandpass` that's under the submodule `filter`.

```
.. _band_pass:

arora.filter.bandpass()
-------------------------
‘‘arora.filter.bandpass(raw_signal, lower_cutoff, upper_cutoff,
    sampling_frequency, filter_order)‘‘

::
Parameters:
    raw_signal : list of int or array like object
        An array with different stages annotated as integers
    lower_cutoff : int or float
        The lower frequency boundary for the bandpass filter.
    upper_cutoff : int or float
        The higher frequency boundary for the bandpass filter
    sampling_frequency : int, default 30
        The frequency in which the raw_signal was sampled in
    filter_order : int
        The order of the filter

Returns: list of float or int
        The array contains the values of signal after having been
    passed through the bandpass filter as ‘‘numpy.ndarray‘‘or array
    like object

For example:
>>> x = [random.randint(5, 10) for _ in range(1200)]
>>> a = arora.filter.bandpass(x, 2, 5, 120, 5)
>>> a
>>> array([ 1.87279319e-05,  1.87031147e-04,  9.27910295e-04, ...,
       -2.73488140e-01, -3.13090143e-01, -3.45810500e-01])
```

Listing 3: rst code excerpt from Arora's documentation

5 Testing

There is a simple system for testing Arora. UnitTests are deployed using Python's built-in unit test framework: `unittest`. Detailed information on the framework can be found at: `https://docs.python.org/3/library/unittest.html`

Basic example

```
import unittest



class MyTestCase(unittest.TestCase):
    def test_something(self):
        self.assertEqual(True, False)  # add assertion here

    def test_something_else(self):
        self.assertEqual(True, False)  # add assertion here


if _name_ == '_main_':
    unittest.main()

```

Listing 4: example of a test file

One test file corresponds to one module file. An IDE such as Visual Studio Code or PyCharm is recommended to run the tests but they can also be executed through the terminal.

To run in an IDE, click the play button next to the function or right-click on the testing file and press "run 'pytest in [filename]'. The same method can be used to run all tests by right-clicking the directory where the tests reside.

If running the tests through the terminal, run the command:

```
pytest path/to/test/file.py
```

Listing 5: running a test file on the terminal

```
pytest path/to/test/file.py::func
```

Listing 6: running a specific function in a test file on the terminal

```
pytest path/to/test/folder
```

Listing 7: running all tests on the terminal

6 Development and Deployment on Gitlab

As part of his final project, Ægir Máni Hauksson developed DevOps functionality of arora. The following text is an excerpt from his README.md file.

6.1 Deployment Command

The recommended deployment method is by the use of this script: `https://gitlab.com/sleep-revolution/sleepy/-/blob/PyPi/scripts/deploy.sh` as it ensures the tag is valid before a pipeline is triggered. If you are on windows or mac, you can execute the script through the git bash terminal. You must have Python 3.8 or greater installed on your machine.

To run the deployment script from within the `/scripts` folder: `bash deploy.sh`

It will stash any current changes you have on your branch, switch to the default branch and prompt you for the required information in the terminal. After completion, your previous git state will be reinstated.

Optionally you can run the command with the required parameters:

```
bash deploy.sh "<new version>" "<tag commit message>" -y
```

You can leave out the -y if you want to confirm your inputs:

```
bash deploy.sh "<new version>" "<tag commit message>"
```

6.2 Semantic Versioning

We follow the semantic versioning convention when deploying our software. The convention holds that an upgrade in MAJOR versions introduces breaking changes that are no longer compatible with earlier versions of the package. MINOR version updates are usually backward compatible changes, and PATCH versions are fixes that are also backward compatible. The easiest way to think of this is to assume that the changes you make are automatically applied to all users of your software unless the update is a MAJOR version update. Thus, any changes you make in MINOR and PATCH versions should not affect the current behavior of an application using an older version of your software after updating.

6.3 Tag Format

The tag should follow the semantic versioning scheme of the format <MAJOR>.<MINOR>.<PATCH>. For example 1.30.24. For the semantic version to be valid in the pipeline, the following conditions must be met. These conditions are validated before a tag commit reaches the git repository (if using the deployment script) and also in the pipeline.

```
1    A new MAJOR version results in a MINOR and PATCH version 0.
2    Before: 1.2.5
3    After: 2.0.0 # VALID - MINOR and PATCH become 0
4
5    Before: 1.2.5
6    After: 2.2.5 # INVALID - MINOR and PATCH not 0
```

```
    A new MINOR version results in a PATCH version 0

    Before: 1.2.5
    After: 1.3.0 # VALID - PATCH becomes 0

    Before: 1.2.5
    After: 1.3.5 # INVALID - PATCH not 0
```

```
    The new MINOR version is one greater than the current latest
    version of the package

    Before: 1.2.5
    After: 1.3.0 # VALID - Version goes from 2 to 3

    Before: 1.2.5
    After: 1.4.0 # INVALID - Version goes from 2 to 4
```

You can remove tags by doing the following

```
    git tag -d <tagname>
    git push --delete origin <tagname>
```

6.4 Deployment Artifacts

The deployed versions of the Arora package produce two artifacts, one of which is the PyPi distribution and the other a tarball within the GitLab package registry.

6.5 PyPi

The Python package is deployed to test.pypi as well as the official pypi website.

Test deployments: `https://test.pypi.org/project/arora/`

Official release: `https://pypi.org/project/arora/`

6.6 Generic Package Registry

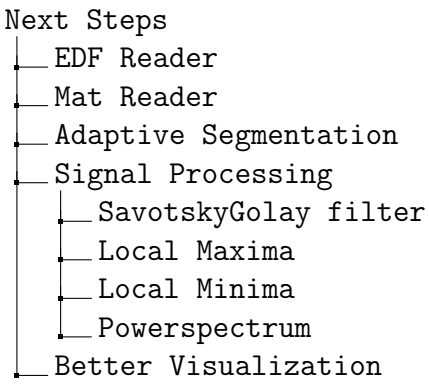
The tarball can be found within the project on Gitlab by navigating to packages registries -> Package Registry in the sidebar of the Gitlab interface. These tarballs can be downloaded directly from the Gitlab interface or fetched via API calls with proper access tokens if needed. These files serve as a backup for each published version of the PyPi distribution. More information on the package API can be found here: `https://docs.gitlab.com/ee/api/packages.html`

7 Roadmap

Due to the structure of this project and the fact that it is in continuous development means that we were not able to achieve every requirement put before us. Here we have outlined the next steps that we believe to matter the most and should be implemented first. This however does not mean that other items will not be implemented in the meantime as the items on the roadmap are meant for the larger items to be implemented or package defining items. These items though are during their implementation process and are subject to change.

7.1 The Next Steps

The items are ranked in the order of what we believe should be implemented/made first.



7.1.1 Improved EDF Reader

The EDF reader that has already been implemented in the package is currently inefficient and impractical for the user. The cluster is unable to handle the RAM usage which makes it hard to test and debug. One solution that has not yet been implemented into Arora is to convert the

EDF files to mat files, with the help of Octave. Python can then read mat files more efficiently.

7.1.2 Reading Matlab Files

As mentioned in 7.1.1 to make a more efficient EDF reader we would be reading the files from mat files. For that, a reader for mat files is needed and there are multiple efficient mat readers available for Python. It would be a good idea at some point to create a new mat reader, that would be more specialized to the wants and needs of the users at Sleep Revolution.

7.1.3 Adaptive Segmentation

The old version of how the signals are segmented has proven to be rather inefficient in explaining the signals and the data itself. A new method for segmentation of sleep data was devised called adaptive segmentation. This new method was different from the previous way of segmenting sleep data in that instead of segmenting data based on some fixed increment, it would find sleep events and create a segment based on the changes in the events. This works by creating a moving window that is continuously compared to a starting window and when the difference is higher than some threshold the segmentation will occur and a new starting window is chosen and then the process repeats itself.

7.1.4 Signal Processing

Signals are the data foundation of what Sleep Revolution is working with and therefore it is important to have many unique features so that the user does not need to use multiple packages to work with the

signal data. Some of the features that came to mind when designing the package but were not implemented due to time constraints are the following;

1. Smoothing the signal. One method that was found was to use the Savotzky-Golay filter to remove those small oscillations and smooth out some of the noise.

2. One item needed to be able to create the Savotzky-Golay filter is finding the local maxima and minima.

3. Finding the amplitude between frequencies can help also with finding noise between signals and also finding the sampling frequency of the signal. This can prove helpful and improve the user experience by no longer requiring the user to remember the sampling frequency of the signal. The solution to this can be creating some analytical power spectrum function that does this work.

7.1.5 Better Visualization

The importance of good graphs and visualization cannot be overstated. Being able to accurately display data in a good and concise way benefits the user when dealing with large amounts of data as EDF files often contain. We were pointed towards Plotly which is an interactive visualization graph that is hosted on a local server and allows the user far more freedom than visualization packages such as matplotlib. As the EDF files contain personal data we were not sure if using a server would be a good idea but seeing as Plotly only creates a local server, available only on your computer we believe that will not pose a problem.