spikeinterface

Jul 31, 2021

Contents:

Spikeinterface is a collection of Python modules designed to improve the accessibility, reliability, and reproducibility of spike sorting and all its associated computations.

With SpikeInterface, users can:

- read/write many extracellular file formats.
- pre-process extracellular recordings.
- run many popular, semi-automatic spike sorters.
- post-process sorted datasets.
- compare and benchmark spike sorting outputs.
- validate and curate spike sorting outputs.
- visualize recordings and spike sorting outputs.

NEWS

- New SpikeInterface release! Version 0.11.0 is now out (on 10/12/2020)!

CHAPTER 1

Overview

Extracellular recordings are an essential source of data in experimental and clinical neuroscience. Of particular interest in these recordings is the activity of single neurons which must be inferred using a blind source separation procedure called spike sorting.

Given the importance of spike sorting, much attention has been directed towards the development of tools and algorithms that can increase its performance and automation. These developments, however, introduce new challenges in software and file format incompatibility which reduce interoperability, hinder benchmarking, and preclude reproducible analysis.

To address these limitations, we developed **SpikeInterface**, a Python framework designed to unify preexisting spike sorting technologies into a single code base and to standardize extracellular data file handling. With a few lines of code, users can run, compare, and benchmark most modern spike sorting algorithms; pre-process, post-process, and visualize extracellular datasets; validate, curate, and export sorted results; and more, regardless of the underlying data format.

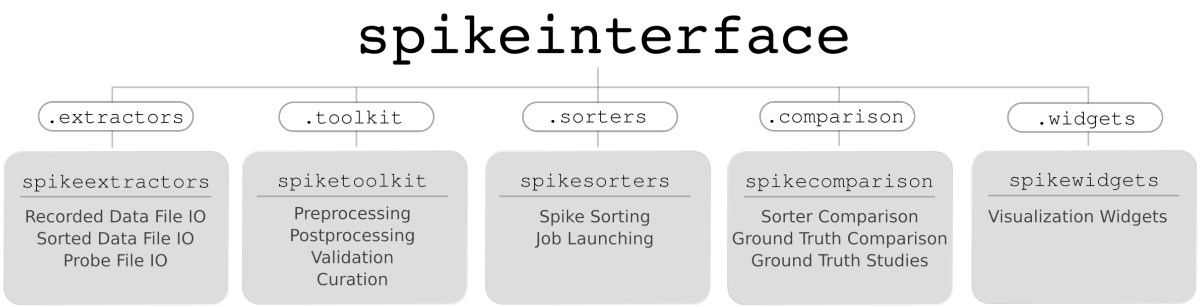
In the following documentation, we provide an overview of SpikeInterface.

1.1 Organization

SpikeInterface consists of 5 main packages which encapsulate all steps in a typical spike sorting pipeline:

- spikeextractors
- spiketoolkit
- spikesorters
- spikecomparison
- spikewidgets

Along with these packages, the `spikeinterface` meta-package allows users to install and use all 5 packages as shown in the figure.



1.2 Related projects

- spikeforest is a reproducible, continuously updating platform which benchmarks the performance of some spike sorting software (kilosort, herdingspike, ironcliust, jrclust, klusta, moutainsort4, spykingcircus, tridesclous, waveclus) using many ground-truth datasets. The processing engine is based on SpikeInterface.
- **probeinterface is a python package to define and handle neural** probes and the wiring to recording devices.
- spikely is a graphical user interface (GUI) that allows users to build and run SpikeInterface spike sorting pipelines on extracellular datasets.
- spikemetrics external python package wrapped by SpikeInterface to compute quality metrics related to spike sorting output.
- spikefeatures external python package wrapped by SpikeInterface to compute different features from extracellular action potentials.
- MEArec is a fast customizable biophysical simulation of extracellular recording.

CHAPTER 2

Installation

`spikeinterface` is a Python package. It can be installed using pip:

```
pip install spikeinterface
```

The pip installation will install a specific and fixed version of the spikeinterface packages.

To use the latest updates, install *spikeinterface* and the related packages from source:

```
git clone https://github.com/SpikeInterface/spikeinterface.git
cd spikeinterface
python setup.py install (or develop)
```

2.1 Requirements

The following Python packages are required for running the full SpikeInterface framework. They are installed when using the pip installer for `spikeinterface`.

- spikeextractors
- spiketoolkit
- spikesorters
- spikecomparison
- spikewidgets

If you installed `spikeinterface` from source, you can install the latest releases of the spikeinterface packages:

```
pip install --upgrade spikeextractors spiketoolkit spikesorters spikecomparison␣
↪spikewidgets
```

You can also install each package from GitHub to keep up with the latest updates. In order to do so, for example for `spikeextractors`, run:

```
pip uninstall spikeextractors
git clone https://github.com/SpikeInterface/spikeextractors
cd spikeextractors
python setup.py install (or develop)
```

CHAPTER 3

Compatible Technology

3.1 Supported File Formats

Currently, we support many popular file formats for both raw and sorted extracellular datasets. Given the standardized, modular design of our recording and sorting extractors, adding new file formats is straightforward so we expect this list to grow in future versions.

We are also integrating extractors based on NEO

3.1.1 Raw Data Formats

For raw data formats, we currently support:

- **BlackRock** - BlackRockRecordingExtractor
- **Binary** - BinDatRecordingExtractor
- **Biocam HDF5** - BiocamRecordingExtractor
- **CED** - CEDRecordingExtractor
- **Experimental Directory Structure (Exdir)** - ExdirRecordingExtractor
- **Intan** - IntanRecordingExtractor
- **Klusta** - KlustaRecordingExtractor
- **MaxOne** - MaxOneRecordingExtractor
- **MCSH5** - MCSH5RecordingExtractor
- **MEArec** - MEArecRecordingExtractor
- **Mountainsort MDA** - MdaRecordingExtractor
- **Neurodata Without Borders** - NwbRecordingExtractor
- **Neuroscope** - NeuroscopeRecordingExtractor

- **NIX** - NIXIORecordingExtractor
- **Neuralynx** - NeuralynxRecordingExtractor
- **Open Ephys** - OpenEphysRecordingExtractor
- **Phy/Kilosort** - PhyRecordingExtractor/KilosortRecordingExtractor
- **Plexon** - PlexonRecordingExtractor
- **Shybrid** - SHYBRIDRecordingExtractor
- **SpikeGLX** - SpikeGLXRecordingExtractor
- **Spyking Circus** - SpykingCircusRecordingExtractor

3.1.2 Sorted Data Formats

For sorted data formats, we currently support:

- **BlackRock** - BlackRockSortingExtractor
- **Combinato** - CombinatoSortingExtractor
- **Cell Explorer** - CellExplorerSortingExtractor
- **Experimental Directory Structure (Exdir)** - ExdirSortingExtractor
- **HerdingSpikes2** - HS2SortingExtractor
- **JRClust** - JRCSortingExtractor
- **Kilosort/Kilosort2** - KiloSortSortingExtractor
- **Klusta** - KlustaSortingExtractor
- **MEArec** - MEArecSortingExtractor
- **Mountainsort MDA** - MdaSortingExtractor
- **Neurodata Without Borders** - NwbSortingExtractor
- **Neuroscope** - NeuroscopeSortingExtractor
- **NPZ (created by SpikeInterface)** - NpzSortingExtractor
- **Open Ephys** - OpenEphysSortingExtractor
- **Shybrid** - SHYBRIDSortingExtractor
- **Spyking Circus** - SpykingCircusSortingExtractor
- **Trideclous** - TridesclousSortingExtractor
- **YASS** - YassSortingExtractor

3.1.3 Installed Extractors

To check which extractors are useable in a given python environment, one can print the installed recording extractor list and the installed sorting extractor list. An example from a newly installed miniconda3 environment is shown below,

First, import the spikeextractors package,

```
import spikeextractors as se
```

Then you can check the installed RecordingExtractor list,

```
se.installed_recording_extractor_list
```

which outputs,

```
[spikeextractors.extractors.mdaextractors.mdaextractors.MdaRecordingExtractor,
 spikeextractors.extractors.biocamrecordingextractor.biocamrecordingextractor.
↪BiocamRecordingExtractor,
 spikeextractors.extractors.bindatrecordingextractor.bindatrecordingextractor.
↪BinDatRecordingExtractor,
 spikeextractors.extractors.spikeglxrecordingextractor.spikeglxrecordingextractor.
↪SpikeGLXRecordingExtractor,
 spikeextractors.extractors.phyextractors.phyextractors.PhyRecordingExtractor,
 spikeextractors.extractors.maxonerecordingextractor.maxonerecordingextractor.
↪MaxOneRecordingExtractor]
```

and the installed SortingExtractors list,

```
se.installed_sorting_extractor_list
```

which outputs,

```
[spikeextractors.extractors.mdaextractors.mdaextractors.MdaSortingExtractor,
 spikeextractors.extractors.hs2sortingextractor.hs2sortingextractor.
↪HS2SortingExtractor,
 spikeextractors.extractors.klustasortingextractor.klustasortingextractor.
↪KlustaSortingExtractor,
 spikeextractors.extractors.kilosortsortingextractor.kilosortsortingextractor.
↪KiloSortSortingExtractor,
 spikeextractors.extractors.phyextractors.phyextractors.PhySortingExtractor,
 spikeextractors.extractors.spykingcircussortingextractor.
↪spykingcircussortingextractor.SpykingCircusSortingExtractor,
 spikeextractors.extractors.npzsortingextractor.npzsortingextractor.
↪NpzSortingExtractor]
```

When trying to use an extractor that has not been installed in your environment, an installation message will appear indicating which python packages must be installed as a prerequisite to using the extractor,

```
exdir_file = 'path_to_exdir_file'
recording = se.ExdirRecordingExtractor(exdir_file)
```

throws the error,

```
----> 1 se.ExdirRecordingExtractor(exdir_file)

~/spikeextractors/spikeextractors/extractors/exdirextractors/exdirextractors.
↪py in __init__(self, exdir_file)
     22
     23     def __init__(self, exdir_file):
---> 24         assert HAVE_EXDIR, "To use the ExdirExtractors run:nn pip␣
↪install exdirnn"
     25         RecordingExtractor.__init__(self)
     26         self._exdir_file = exdir_file
```

```
AssertionError: To use the ExdirExtractors run:

pip install exdir
```

So to use either of the Exdir extractors, you must install the python package exdir. The python packages that are required to use of all the extractors can be installed as below,

```
pip install exdir h5py pyintan MEArec pyopenephys tridesclous
```

3.2 Dealing with Non-Supported File Formats

Many users store their datasets in custom file formats that are not general enough to create new extractors. To allow these users to still utilize SpikeInterface with their data, we built two in-memory Extractors: the **NumpyRecordingExtractor** and the **NumpySortingExtractor**.

The NumpyRecordingExtractor can be instantiated with a numpy array that contains the underlying extracellular traces (channels x frames), the sampling frequency, and the probe geometry (optional). Once instantiated, the NumpyRecordingExtractor can be used like any other RecordingExtractor.

The NumpySortingExtractor does not need any data during instantiation. However, after instantiation, it can be filled with data using its built-in functions (load_from_extractor, set_times_labels, and add_unit). After sorted data is added to the NumpySortingExtractor, it can be used like any other SortingExtractor.

With these two objects, we hope that any user can access SpikeInterface regardless of the nature of their underlying file format. If you feel like a non-supported file format should be included in SpikeInterface as an actual extractor, please leave an issue in the spikeextractors repository.

3.3 Supported Spike Sorters

Currently, we support many popular semi-automatic spike sorters. Given the standardized, modular design of our sorters, adding new ones is straightforward so we expect this list to grow in future versions.

- **HerdingSpikes2** - HerdingspikesSorter
- **IronClust** - IronClustSorter
- **Kilosort** - KilosortSorter
- **Kilosort2** - Kilosort2Sorter
- **Klusta** - KlustaSorter
- **Mountainsort4** - Mountainsort4Sorter
- **SpyKING Circus** - SpykingcircusSorter
- **Tridesclous** - TridesclousSorter
- **Wave clus** - WaveClusSorter

3.3.1 Installed Sorters

To check which sorters are useable in a given python environment, one can print the installed sorters list. An example is shown in a pre-defined miniconda3 environment.

First, import the spikesorters package,

```
import spikesorters as ss
```

Then you can check the installed Sorter list,

```
ss.installed_sorters()
```

which outputs,

```
['herdingspikes',
 'klusta',
 'mountainsort4',
 'spykingcircus',
 'tridesclous']
```

When trying to use an sorter that has not been installed in your environment, an installation message will appear indicating how to install the given sorter,

```
recording = sorters.run_ironclust(recording)
```

throws the error,

```
AssertionError: This sorter ironclust is not installed.
      Please install it with:

To use IronClust run:

      >>> git clone https://github.com/jamesjun/ironclust
  and provide the installation path by setting the IRONCLUST_PATH
  environment variables or using IronClustSorter.set_ironclust_path().
```

CHAPTER 4

Installing Spike Sorters

An important aspect of spikeinterface is the *spikeinterface.sorters* module. This module wraps many popular spike sorting tools. This means that you can run multiple sorters on the same dataset with only a few lines of code and through Python.

These spike sorting algorithms **must be installed externally**. Some of theses sorters are written in Matlab, so you will also to install Matlab if you want to use them (Kilosort, Kilosort2, Ironclust, ...) Some of then will also need some computing library like CUDA (Kilosort, Kilosort2, Ironclust (optional)) or opencl (Tridesclous) to use hardware acceleration (GPU).

Here is a list of the implemented wrappers and some instructions to install them on your local machine. Installation instructions are given for an **Unbuntu** platform. Please check the documentation of the different spike sorters to retrieve installation instructions for other operating systems. We use **pip** to install packages, but **conda** should also work in many cases.

If you experience installation problems please directly contact the authors of theses tools or write on the related mailing list, google group, etc.

Please feel free to enhance this document with more installation tips.

4.1 Herdingspikes2

- Python + C++
- Url: https://github.com/mhhennig/hs2
- Authors: Matthias Hennig, Jano Horvath,Cole Hurwitz, Oliver Muthmann, Albert Puente Encinas, Martino Sorbaro, Cesar Juarez Ramirez, Raimon Wintzer: GUI and visualisation
- Installation:

```
pip install herdingspikes
```

4.2 HDSort

- Matlab
- Url: https://git.bsse.ethz.ch/hima_public/HDsort.git
- Authors: Roland Diggelmann, Felix Franke
- Installation:

```
git clone https://git.bsse.ethz.ch/hima_public/HDsort.git
# provide installation path by setting the HDSORT_PATH environment variable
# or using HDSortSorter.set_hdsort_path()
```

4.3 IronClust

- Matlab
- Url: https://github.com/jamesjun/ironclust
- Authors: James J. Jun
- Installation need Matlab:

```
git clone https://github.com/jamesjun/ironclust
# provide installation path by setting the IRONCLUST_PATH environment variable
# or using IronClustSorter.set_ironclust_path()
```

4.4 Kilosort

- Matlab, requires CUDA
- Url: https://github.com/cortex-lab/KiloSort
- Authors: Marius Pachitariu
- Installation needs Matlab and cudatoolkit:

```
git clone https://github.com/cortex-lab/KiloSort
# provide installation path by setting the KILOSORT_PATH environment variable
# or using KilosortSorter.set_kilosort_path()
```

- See also for Matlab/cuda: https://www.mathworks.com/help/parallel-computing/gpu-support-by-release.html

4.5 Kilosort2

- Matlab, requires CUDA
- Url: https://github.com/MouseLand/Kilosort2
- Authors: Marius Pachitariu
- Installation needs Matlab and cudatoolkit:

```
git clone https://github.com/MouseLand/Kilosort2
# provide installation path by setting the KILOSORT2_PATH environment variable
# or using Kilosort2Sorter.set_kilosort2_path()
```

- See also for Matlab/cuda: https://www.mathworks.com/help/parallel-computing/gpu-support-by-release.html

4.6 Kilosort2.5

- Matlab, requires CUDA
- Url: https://github.com/MouseLand/Kilosort
- Authors: Marius Pachitariu
- Installation needs Matlab and cudatoolkit:

```
git clone https://github.com/MouseLand/Kilosort
# provide installation path by setting the KILOSORT2_5_PATH environment variable
# or using Kilosort2_5Sorter.set_kilosort2_path()
```

- See also for Matlab/cuda: https://www.mathworks.com/help/parallel-computing/gpu-support-by-release.html

4.7 Klusta

- Python
- Url: https://github.com/kwikteam/klusta
- Authors: Cyrille Rossant, Shabnam Kadir, Dan Goodman, Max Hunter, Kenneth Harris
- Installation:

```
pip install Cython h5py tqdm
pip install click klusta klustakwik2
```

- See also: https://github.com/kwikteam/phy

4.8 Mountainsort4

- Python
- Url: https://github.com/flatironinstitute/mountainsort
- Authors: Jeremy Magland, Alex Barnett, Jason Chung, Loren Frank, Leslie Greengard
- Installation:

```
pip install ml_ms4alg
```

4.9 SpykingCircus

- Python, requires MPICH
- Url: https://spyking-circus.readthedocs.io
- Authors: Pierre Yger, Olivier Marre
- Installation:

```
sudo apt install libmpich-dev
pip install mpi4py
pip install spyking-circus --no-binary=mpi4py
```

4.10 Tridesclous

- Python, runs faster with opencl installed but optional
- Url: https://tridesclous.readthedocs.io
- Authors: Samuel Garcia, Christophe Pouzat
- Installation:

```
pip install tridesclous
```

- Optional installation of opencl ICD and pyopencl for hardware acceleration:

```
sudo apt-get install beignet (optional if intel GPU)
sudo apt-get install nvidia-opencl-XXX (optional if nvidia GPU)
sudo apt-get install pocl-opencl-icd (optional for multi core CPU)
sudo apt-get install opencl-headers ocl-icd-opencl-dev libclc-dev ocl-icd-
↪libopencl1
pip install pyopencl
```

4.11 Waveclus

- Matlab
- Url: https://github.com/csn-le/wave_clus/wiki
- Authors: Fernando Chaure, Hernan Rey and Rodrigo Quian Quiroga
- Installation needs Matlab:

```
git clone https://github.com/csn-le/wave_clus/
# provide installation path by setting the WAVECLUS_PATH environment variable
# or using WaveClusSorter.set_waveclus_path()
```

4.12 Combinato

- Python
- Url: https://github.com/jniediek/combinato/wiki

- Authors: Johannes Niediek, Jan Boström, Christian E. Elger, Florian Mormann
- Installation:

```
git clone https://github.com/jniediek/combinato
# Then inside that folder, run:
python setup_options.py
# provide installation path by setting the COMBINATO_PATH environment variable
# or using CombinatoSorter.set_combinato_path()
```

CHAPTER 5

Getting started with SpikeInterface

In this introductory example, you will see how to use the `spikeinterface` to perform a full electrophysiology analysis. We will first create some simulated data, and we will then perform some pre-processing, run a couple of spike sorting algorithms, inspect and validate the results, export to Phy, and compare spike sorters.

Let's first import the `spikeinterface` package. We can either import the whole package:

```
import spikeinterface as si
```

or import the different submodules separately (preferred). There are 5 modules which correspond to 5 separate packages:

- `extractors` : file IO and probe handling
- `toolkit` : processing toolkit for pre-, post-processing, validation, and automatic curation
- `sorters` : Python wrappers of spike sorters
- `comparison` : comparison of spike sorting output
- `widgets` : visualization

```
import spikeinterface.extractors as se
import spikeinterface.toolkit as st
import spikeinterface.sorters as ss
import spikeinterface.comparison as sc
import spikeinterface.widgets as sw
```

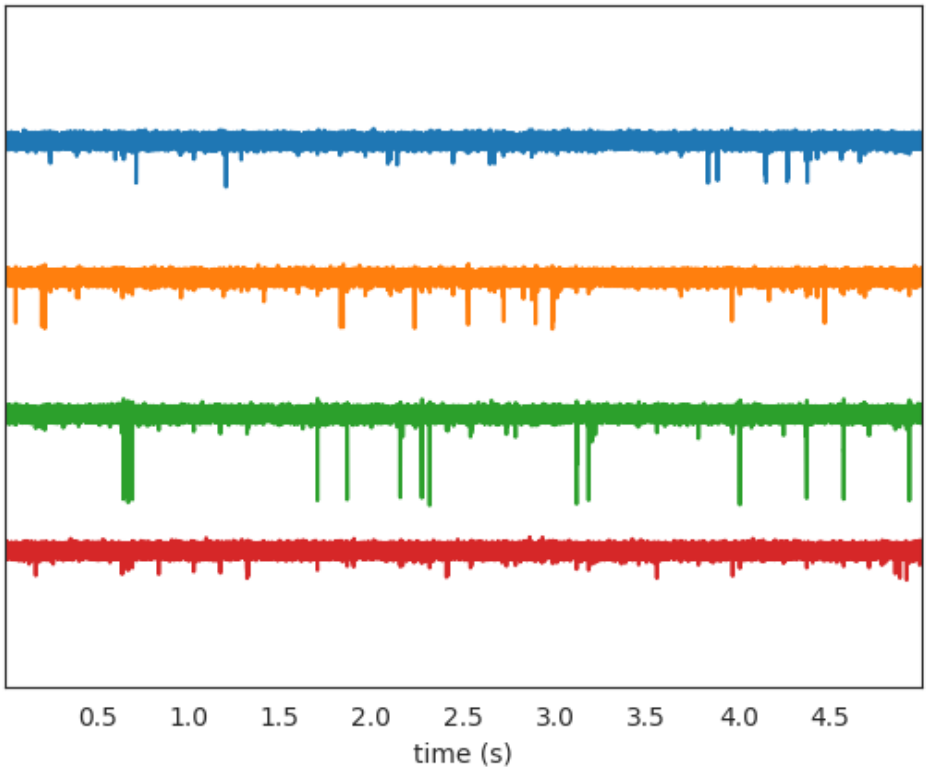
First, let's create a toy example with the `extractors` module:

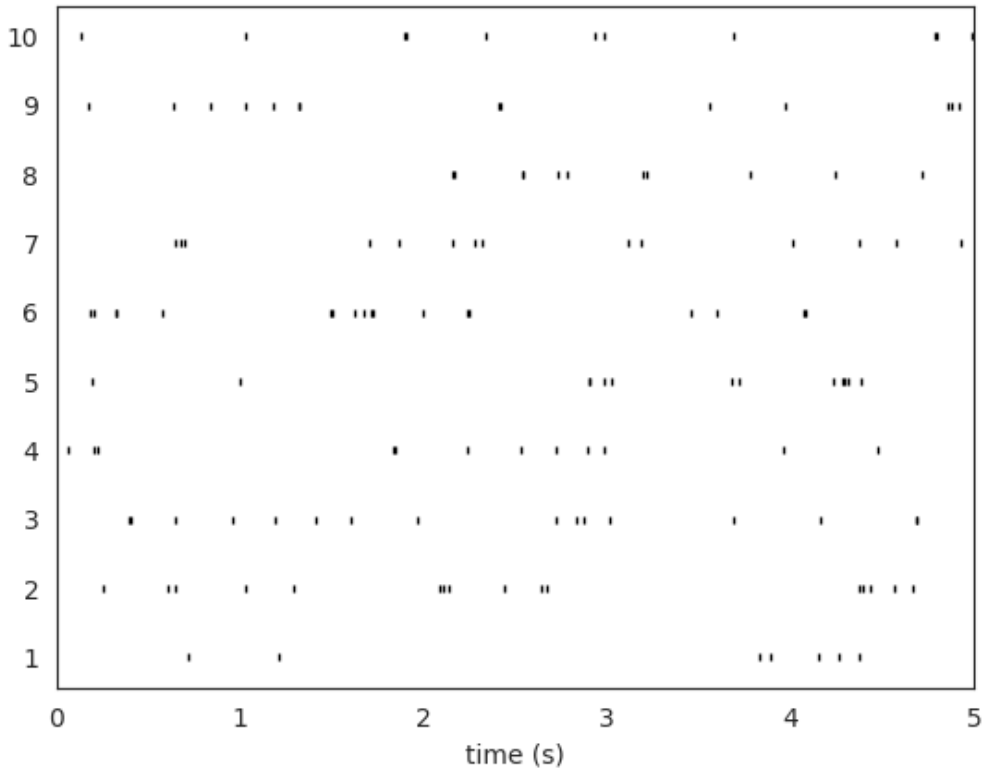
```
recording, sorting_true = se.example_datasets.toy_example(duration=10, num_channels=4,
↪ seed=0)
```

`recording` is a `RecordingExtractor` object, which extracts information about channel ids, channel locations (if present), the sampling frequency of the recording, and the extracellular traces. `sorting_true` is a `SortingExtractor` object, which contains information about spike-sorting related information, including unit ids, spike trains, etc. Since the data are simulated, `sorting_true` has ground-truth information of the spiking activity of each unit.

Let's use the `widgets` module to visualize the traces and the raster plots.

```
w_ts = sw.plot_timeseries(recording, trange=[0,5])
w_rs = sw.plot_rasters(sorting_true, trange=[0,5])
```





-

This is how you retrieve info from a `RecordingExtractor`...

```
channel_ids = recording.get_channel_ids()
fs = recording.get_sampling_frequency()
num_chan = recording.get_num_channels()

print('Channel ids:', channel_ids)
print('Sampling frequency:', fs)
print('Number of channels:', num_chan)
```

Out:

```
Channel ids: [0, 1, 2, 3]
Sampling frequency: 30000.0
Number of channels: 4
```

...and a `SortingExtractor`

```
unit_ids = sorting_true.get_unit_ids()
spike_train = sorting_true.get_unit_spike_train(unit_id=unit_ids[0])

print('Unit ids:', unit_ids)
print('Spike train of first unit:', spike_train)
```

Out:

```
Unit ids: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Spike train of first unit: [ 21478  36186 115033 116600 124535 127993 131277 159400␣
↪163465 164645
 183164 192248 196081 214557 215749 233858 234350 250897 263249 267532
 284621 293768]
```

Optionally, you can load probe information using a '.prb' file. For example, this is the content of `custom_probe.prb`:

```
channel_groups = {
    0: {
        'channels': [1, 0],
        'geometry': [[0, 0], [0, 1]],
        'label': ['first_channel', 'second_channel'],
    },
    1: {
        'channels': [2, 3],
        'geometry': [[3,0], [3,1]],
        'label': ['third_channel', 'fourth_channel'],
    }
}
```

The '.prb' file uses python-dictionary syntax. With probe files you can change the order of the channels, load 'group' properties, 'location' properties (using the 'geometry' or 'location' keys, and any other arbitrary information (e.g. 'labels'). All information can be specified as lists (same number of elements of corresponding 'channels' in 'channel_group', or dictionaries with the channel id as key and the property as value (e.g. 'labels': {1: 'first_channel', 0: 'second_channel'})

You can load the probe file using the `load_probe_file` function in the RecordingExtractor. **IMPORTANT**: The `load_probe_file` function returns a ***new** `RecordingExtractor` object and it is not performed in-place:

```
recording_prb = recording.load_probe_file('custom_probe.prb')
print('Channel ids:', recording_prb.get_channel_ids())
print('Loaded properties', recording_prb.get_shared_channel_property_names())
print('Label of channel 0:', recording_prb.get_channel_property(channel_id=0,␣
↪property_name='label'))

# 'group' and 'location' can be returned as lists:
print(recording_prb.get_channel_groups())
print(recording_prb.get_channel_locations())
```

Out:

```
Channel ids: [1, 0, 2, 3]
Loaded properties ['gain', 'group', 'label', 'location', 'offset']
Label of channel 0: second_channel
[0 0 1 1]
[[0. 0.]
 [0. 1.]
 [3. 0.]
 [3. 1.]]
```

Using the `toolkit`, you can perform pre-processing on the recordings. Each pre-processing function also returns a `RecordingExtractor`, which makes it easy to build pipelines. Here, we filter the recording and apply common median reference (CMR)

```
recording_f = st.preprocessing.bandpass_filter(recording, freq_min=300, freq_max=6000)
recording_cmr = st.preprocessing.common_reference(recording_f, reference='median')
```

Now you are ready to spikesort using the `sorters` module! Let's first check which sorters are implemented and which are installed

```
print('Available sorters', ss.available_sorters())
print('Installed sorters', ss.installed_sorters())
```

Out:

```
Available sorters ['combinato', 'hdsort', 'herdingspikes', 'ironclust', 'kilosort',
↪'kilosort2', 'kilosort2_5', 'kilosort3', 'klusta', 'mountainsort4', 'spykingcircus',
↪ 'tridesclous', 'waveclus', 'yass']
Installed sorters ['klusta', 'mountainsort4', 'tridesclous']
```

The `ss.installed_sorters()` will list the sorters installed in the machine. We can see we have Klusta and Mountainsort4 installed. Spike sorters come with a set of parameters that users can change. The available parameters are dictionaries and can be accessed with:

```
print(ss.get_default_params('mountainsort4'))
print(ss.get_default_params('klusta'))
```

Out:

```
{'detect_sign': -1, 'adjacency_radius': -1, 'freq_min': 300, 'freq_max': 6000, 'filter
↪': True, 'whiten': True, 'curation': False, 'num_workers': None, 'clip_size': 50,
↪'detect_threshold': 3, 'detect_interval': 10, 'noise_overlap_threshold': 0.15}
{'adjacency_radius': None, 'threshold_strong_std_factor': 5, 'threshold_weak_std_
↪factor': 2, 'detect_sign': -1, 'extract_s_before': 16, 'extract_s_after': 32, 'n_
↪features_per_channel': 3, 'pca_n_waveforms_max': 10000, 'num_starting_clusters': 50,
↪ 'chunk_mb': 500, 'n_jobs_bin': 1}
```

Let's run mountainsort4 and change one of the parameter, the detection_threshold:

```
sorting_MS4 = ss.run_mountainsort4(recording=recording_cmr, detect_threshold=6)
```

Out:

```
Warning! The recording is already filtered, but Mountainsort4 filter is enabled. You␣
↪can disable filters by setting 'filter' parameter to False
```

Alternatively we can pass full dictionary containing the parameters:

```
ms4_params = ss.get_default_params('mountainsort4')
ms4_params['detect_threshold'] = 4
ms4_params['curation'] = False

# parameters set by params dictionary
sorting_MS4_2 = ss.run_mountainsort4(recording=recording, **ms4_params)
```

Out:

```
Warning! The recording is already filtered, but Mountainsort4 filter is enabled. You␣
↪can disable filters by setting 'filter' parameter to False
```

Let's run Klusta as well, with default parameters:

```
sorting_KL = ss.run_klusta(recording=recording_cmr)
```

Out:

```
RUNNING SHELL SCRIPT: /home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/
↪checkouts/0.13.0/examples/getting_started/klusta_output/run_klusta.sh
/home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/checkouts/0.13.0/doc/
↪sources/spikesorters/spikesorters/basesorter.py:158: ResourceWarning: unclosed file
↪<_io.TextIOWrapper name=6 encoding='UTF-8'>
  self._run(recording, self.output_folders[i])
```

The `sorting_MS4` and `sorting_MS4` are `SortingExtractor` objects. We can print the units found using:

```
print('Units found by Mountainsort4:', sorting_MS4.get_unit_ids())
print('Units found by Klusta:', sorting_KL.get_unit_ids())
```

Out:

```
Units found by Mountainsort4: [1, 2, 3, 4, 5, 6]
Units found by Klusta: [0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

Once we have paired `RecordingExtractor` and `SortingExtractor` objects we can post-process, validate, and curate the results. With the `toolkit.postprocessing` submodule, one can, for example, get waveforms, templates, maximum channels, PCA scores, or export the data to Phy. Phy is a GUI for manual curation of the spike sorting output. To export to phy you can run:

```
st.postprocessing.export_to_phy(recording, sorting_KL, output_folder='phy')
```

Then you can run the template-gui with: `phy template-gui phy/params.py` and manually curate the results.

Validation of spike sorting output is very important. The `toolkit.validation` module implements several quality metrics to assess the goodness of sorted units. Among those, for example, are signal-to-noise ratio, ISI violation ratio, isolation distance, and many more.

```
snrs = st.validation.compute_snrs(sorting_KL, recording_cmr)
isi_violations = st.validation.compute_isi_violations(sorting_KL, duration_in_
↪frames=recording_cmr.get_num_frames())
isolations = st.validation.compute_isolation_distances(sorting_KL, recording_cmr)

print('SNR', snrs)
print('ISI violation ratios', isi_violations)
print('Isolation distances', isolations)
```

Out:

```
SNR [42.46399044 65.03311047 33.47233692  4.08219477  3.85108472 15.98982149
 30.0173599  13.51226553  3.90097428 10.24819052 11.55758006  4.07045108]
ISI violation ratios [0.         0.         0.         0.54469044 0.40498685 2.
↪60091552
 0.         7.4906367  0.39824982 2.10674157 0.         0.45861041]
Isolation distances [           nan 2.05579984e+04 1.82982173e+03 6.58686067e+00
 6.11073398e+00 1.98799187e+01 1.67309619e+03 9.72017815e+00
 8.77673461e+00 1.14200866e+01 1.31270206e+01 9.90547581e+00]
```

Quality metrics can be also used to automatically curate the spike sorting output. For example, you can select sorted units with a SNR above a certain threshold:

```
sorting_curated_snr = st.curation.threshold_snrs(sorting_KL, recording_cmr,␣
↪threshold=5, threshold_sign='less')
snrs_above = st.validation.compute_snrs(sorting_curated_snr, recording_cmr)

print('Curated SNR', snrs_above)
```

Out:

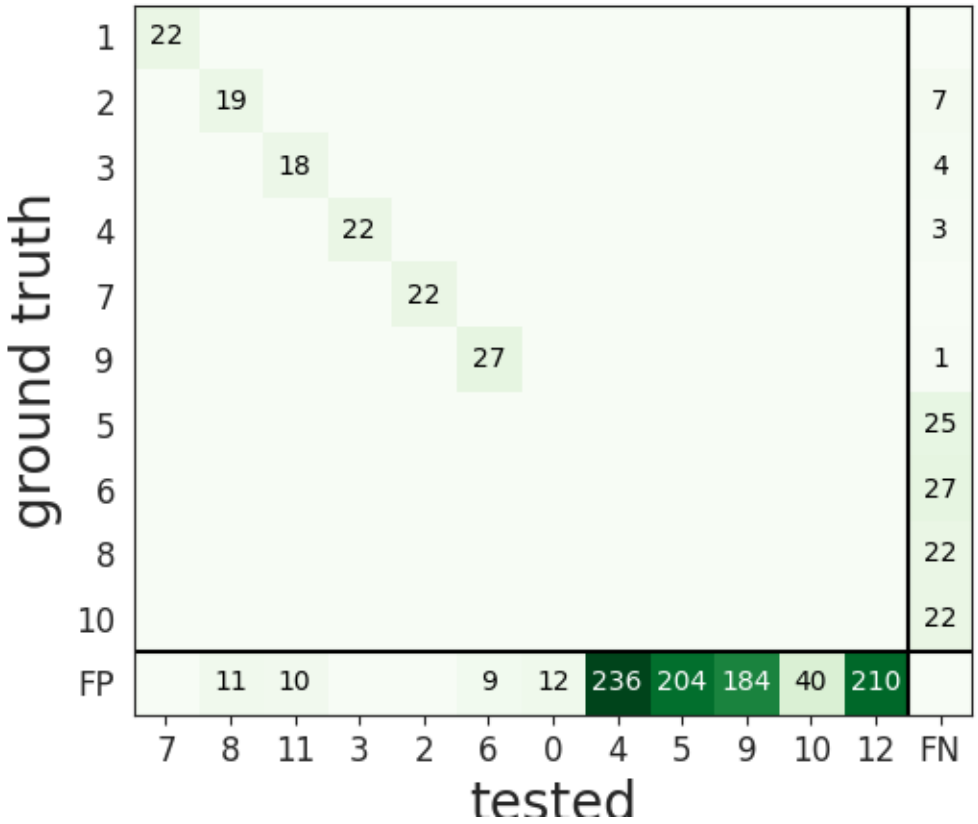
```
Curated SNR [42.46399044 65.03311047 33.47233692 15.98982149 30.0173599  13.51226553
 10.24819052 11.55758006]
```

The final part of this tutorial deals with comparing spike sorting outputs. We can either (1) compare the spike sorting results with the ground-truth sorting `sorting_true`, (2) compare the output of two (Klusta and Mountainsor4), or (3) compare the output of multiple sorters:

```
comp_gt_KL = sc.compare_sorter_to_ground_truth(gt_sorting=sorting_true, tested_
↪sorting=sorting_KL)
comp_KL_MS4 = sc.compare_two_sorters(sorting1=sorting_KL, sorting2=sorting_MS4)
comp_multi = sc.compare_multiple_sorters(sorting_list=[sorting_MS4, sorting_KL],
                                         name_list=['klusta', 'ms4'])
```

When comparing with a ground-truth sorting extractor (1), you can get the sorting performance and plot a confusion matrix

```
comp_gt_KL.get_performance()
w_conf = sw.plot_confusion_matrix(comp_gt_KL)
```



When comparing two sorters (2), we can see the matching of units between sorters. For example, this is how to extract the unit ids of Mountainsort4 (sorting2) mapped to the units of Klusta (sorting1). Units which are not mapped has -1 as unit id.

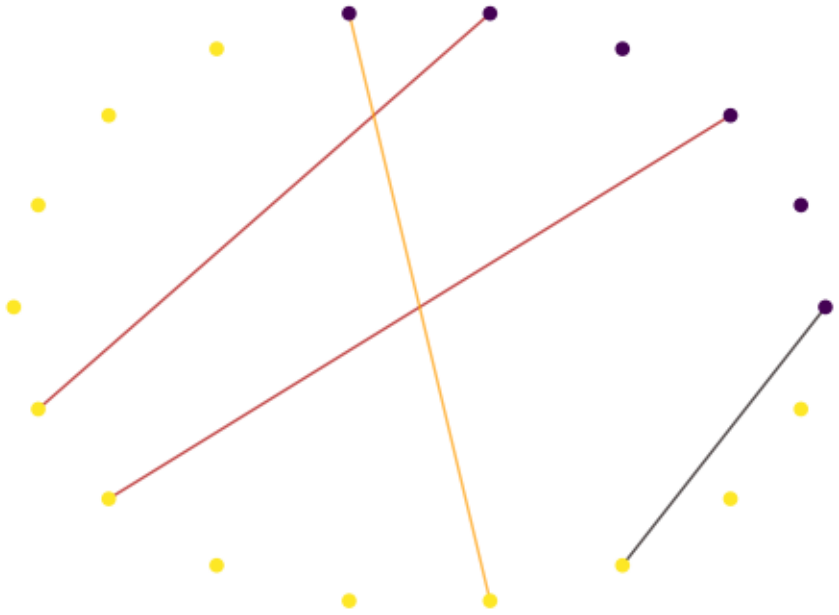
```
mapped_units = comp_KL_MS4.get_mapped_sorting1().get_mapped_unit_ids()

print('Klusta units:', sorting_KL.get_unit_ids())
print('Mapped Mountainsort4 units:', mapped_units)
```

Out:

```
Klusta units: [0, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
Mapped Mountainsort4 units: [-1, 5, 3, -1, -1, 6, 1, -1, -1, -1, -1, -1]
```

When comparing multiple sorters (3), you can extract a `SortingExtractor` object with units in agreement between sorters. You can also plot a graph showing how the units are matched between the sorters.

```
sorting_agreement = comp_multi.get_agreement_sorting(minimum_agreement_count=2)

print('Units in agreement between Klusta and Mountainsort4:', sorting_agreement.get_
↪unit_ids())

w_multi = sw.plot_multicomp_graph(comp_multi)
```



Out:

```
Units in agreement between Klusta and Mountainsort4: [0, 2, 4, 5]
```

Total running time of the script: (0 minutes 18.281 seconds)

CHAPTER 6

Tutorials

Spike interface is split in 5 modules. Here are tutorials for each one.

6.1 Extractors tutorials

The `extractors` module imports the spikeextractors package. It is designed to load and save recorded and sorted data and to handle probe information.

- RecordingExtractor
- SortingExtractor
- Handling probe information

6.1.1 RecordingExtractor objects

The `RecordingExtractor` is the basic class for handling recorded data. Here is how it works.

```
import numpy as np
import spikeinterface.extractors as se
```

We will create a `RecordingExtractor` object from scratch using `numpy` and the `NumpyRecordingExtractor`

Let's define the properties of the dataset

```
num_channels = 7
sampling_frequency = 30000  # in Hz
duration = 20
num_timepoints = int(sampling_frequency * duration)
```

We can generate a pure-noise timeseries dataset recorded by a linear probe geometry

```
timeseries = np.random.normal(0, 10, (num_channels, num_timepoints))
geom = np.zeros((num_channels, 2))
geom[:, 0] = range(num_channels)
```

And instantiate a `NumpyRecordingExtractor`:

```
recording = se.NumpyRecordingExtractor(timeseries=timeseries, geom=geom, sampling_
↪frequency=sampling_frequency)
```

We can now print properties that the `RecordingExtractor` retrieves from the underlying recording.

```
print('Num. channels = {}'.format(len(recording.get_channel_ids())))
print('Sampling frequency = {} Hz'.format(recording.get_sampling_frequency()))
print('Num. timepoints = {}'.format(recording.get_num_frames()))
print('Stdev. on third channel = {}'.format(np.std(recording.get_traces(channel_
↪ids=2))))
print('Location of third electrode = {}'.format(recording.get_channel_
↪property(channel_id=2, property_name='location')))
```

Out:

```
Num. channels = 7
Sampling frequency = 30000.0 Hz
Num. timepoints = 600000
Stdev. on third channel = 9.993832082475272
Location of third electrode = [2. 0.]
```

Some extractors also implement a `write` function. We can for example save our newly created recording into MDA format (Mountainsort4 format):

```
se.MdaRecordingExtractor.write_recording(recording=recording, save_path='sample_
↪mountainsort_dataset')
```

and read it back with the proper extractor:

```
recording2 = se.MdaRecordingExtractor(folder_path='sample_mountainsort_dataset')
print('Num. channels = {}'.format(len(recording2.get_channel_ids())))
print('Sampling frequency = {} Hz'.format(recording2.get_sampling_frequency()))
print('Num. timepoints = {}'.format(recording2.get_num_frames()))
print('Stdev. on third channel = {}'.format(np.std(recording2.get_traces(channel_
↪ids=2))))
print('Location of third electrode = {}'.format(recording.get_channel_
↪property(channel_id=2, property_name='location')))
```

Out:

```
Num. channels = 7
Sampling frequency = 30000.0 Hz
Num. timepoints = 600000
Stdev. on third channel = 9.993831634521484
Location of third electrode = [2. 0.]
```

Sometimes experiments are run with different conditions, e.g. a drug is applied, or stimulation is performed. In order to define different phases of an experiment, one can use epochs:

```
recording2.add_epoch(epoch_name='stimulation', start_frame=1000, end_frame=6000)
recording2.add_epoch(epoch_name='post_stimulation', start_frame=6000, end_frame=10000)
```

(continues on next page)

(continued from previous page)

```
recording2.add_epoch(epoch_name='pre_stimulation', start_frame=0, end_frame=1000)

recording2.get_epoch_names()
```

Out:

```
['pre_stimulation', 'stimulation', 'post_stimulation']
```

An Epoch can be retrieved and it is returned as a `SubRecordingExtractor`, which is a subclass of the `RecordingExtractor`, hence maintaining the same functionality.

```
recording3 = recording2.get_epoch(epoch_name='stimulation')
epoch_info = recording2.get_epoch_info('stimulation')
start_frame = epoch_info['start_frame']
end_frame = epoch_info['end_frame']

print('Epoch Name = stimulation')
print('Start Frame = {}'.format(start_frame))
print('End Frame = {}'.format(end_frame))
print('Mean. on second channel during stimulation = {}'.format(np.mean(recording3.get_
↪traces(channel_ids=1))))
print('Location of third electrode = {}'.format(recording.get_channel_
↪property(channel_id=2, property_name='location')))
```

Out:

```
Epoch Name = stimulation
Start Frame = 1000
End Frame = 6000
Mean. on second channel during stimulation = 0.22974658012390137
Location of third electrode = [2. 0.]
```

`SubRecordingExtractor` objects can be used to extract arbitrary subsets of your data/channels manually without epoch functionality:

```
recording4 = se.SubRecordingExtractor(parent_recording=recording2, channel_ids=[2, 3,␣
↪4, 5], start_frame=14000,
                                      end_frame=16000)

print('Num. channels = {}'.format(len(recording4.get_channel_ids())))
print('Sampling frequency = {} Hz'.format(recording4.get_sampling_frequency()))
print('Num. timepoints = {}'.format(recording4.get_num_frames()))
print('Stdev. on third channel = {}'.format(np.std(recording4.get_traces(channel_
↪ids=2))))
print(
    'Location of third electrode = {}'.format(recording4.get_channel_property(channel_
↪id=2, property_name='location')))
```

Out:

```
Num. channels = 4
Sampling frequency = 30000.0 Hz
Num. timepoints = 2000
Stdev. on third channel = 9.710132598876953
Location of third electrode = [2. 0.]
```

or to remap the channel ids:

```
recording5 = se.SubRecordingExtractor(parent_recording=recording2, channel_ids=[2, 3,␣
↪4, 5],
                                      renamed_channel_ids=[0, 1, 2, 3],
                                      start_frame=14000, end_frame=16000)
print('New ids = {}'.format(recording5.get_channel_ids()))
print('Original ids = {}'.format(recording5.get_original_channel_ids([0, 1, 2, 3])))
print('Num. channels = {}'.format(len(recording5.get_channel_ids())))
print('Sampling frequency = {} Hz'.format(recording5.get_sampling_frequency()))
print('Num. timepoints = {}'.format(recording5.get_num_frames()))
print('Stdev. on third channel = {}'.format(np.std(recording5.get_traces(channel_
↪ids=0))))
print(
    'Location of third electrode = {}'.format(recording5.get_channel_property(channel_
↪id=0, property_name='location')))
```

Out:

```
New ids = [0, 1, 2, 3]
Original ids = [2, 3, 4, 5]
Num. channels = 4
Sampling frequency = 30000.0 Hz
Num. timepoints = 2000
Stdev. on third channel = 9.710132598876953
Location of third electrode = [2. 0.]
```

Total running time of the script: (0 minutes 0.257 seconds)

6.1.2 SortingExtractor objects

The `SortingExtractor` is the basic class for handling spike sorted data. Here is how it works.

```
import numpy as np
import spikeinterface.extractors as se
```

We will create a `SortingExtractor` object from scratch using `numpy` and the `NumpySortingExtractor`

Let's define the properties of the dataset

```
sampling_frequency = 30000
duration = 20
num_timepoints = int(sampling_frequency * duration)
num_units = 4
num_events = 1000
```

We generate some random events.

```
times = np.int_(np.sort(np.random.uniform(0, num_timepoints, num_events)))
labels = np.random.randint(1, num_units + 1, size=num_events)
```

And instantiate a `NumpyRecordingExtractor`:

```
sorting = se.NumpySortingExtractor()
sorting.set_times_labels(times=times, labels=labels)
sorting.set_sampling_frequency(sampling_frequency=sampling_frequency)
```

We can now print properties that the `SortingExtractor` retrieves from the underlying sorted dataset.

```
print('Unit ids = {}'.format(sorting.get_unit_ids()))
st = sorting.get_unit_spike_train(unit_id=1)
print('Num. events for unit 1 = {}'.format(len(st)))
st1 = sorting.get_unit_spike_train(unit_id=1, start_frame=0, end_frame=30000)
print('Num. events for first second of unit 1 = {}'.format(len(st1)))
```

Out:

```
Unit ids = [1, 2, 3, 4]
Num. events for unit 1 = 267
Num. events for first second of unit 1 = 8
```

Some extractors also implement a `write` function. We can for example save our newly created sorting into MDA format (Mountainsort4 format):

```
se.MdaSortingExtractor.write_sorting(sorting=sorting, save_path='firings_true.mda')
```

and read it back with the proper extractor:

```
sorting2 = se.MdaSortingExtractor(file_path='firings_true.mda',
                                  sampling_frequency=sampling_frequency)
print('Unit ids = {}'.format(sorting2.get_unit_ids()))
st = sorting2.get_unit_spike_train(unit_id=1)
print('Num. events for unit 1 = {}'.format(len(st)))
st1 = sorting2.get_unit_spike_train(unit_id=1, start_frame=0, end_frame=30000)
print('Num. events for first second of unit 1 = {}'.format(len(st1)))
```

Out:

```
Unit ids = [1, 2, 3, 4]
Num. events for unit 1 = 267
Num. events for first second of unit 1 = 8
```

Unit properties are name value pairs that we can store for any unit. We will now calculate a unit property and store it in the `SortingExtractor`

```
full_spike_train = sorting2.get_unit_spike_train(unit_id=1)
firing_rate = float(len(full_spike_train)) / duration
sorting2.set_unit_property(unit_id=1, property_name='firing_rate', value=firing_rate)
print('Average firing rate during the recording of unit 1 = {}'.format(sorting2.get_
↪unit_property(unit_id=1,
                                                                                    ⌋
↪            property_name=
                                                                                    ⌋
↪            'firing_rate')))
print("Spike property names: " + str(sorting2.get_unit_property_names(unit_id=1)))
```

Out:

```
Average firing rate during the recording of unit 1 = 13.35
Spike property names: ['firing_rate']
```

`SubSortingExtractor` objects can be used to extract arbitrary subsets of your units/spike trains manually

```
sorting3 = se.SubSortingExtractor(parent_sorting=sorting2, unit_ids=[1, 2],
                                  start_frame=10000, end_frame=20000)
print('Num. units = {}'.format(len(sorting3.get_unit_ids())))
```

(continues on next page)

(continued from previous page)

```
print('Average firing rate of units1 during frames 10000-20000 = {}'.format(
    float(len(sorting3.get_unit_spike_train(unit_id=1))) / (10000 / sorting3.get_
↪sampling_frequency())))
```

Out:

```
Num. units = 2
Average firing rate of units1 during frames 10000-20000 = 3.0
```

Unit features are name value pairs that we can store for each spike. Let's load a randomly generated 'random_value' features. Features are used, for example, to store waveforms, amplitude, and PCA scores

```
random_values = np.random.randn(len(sorting3.get_unit_spike_train(unit_id=1)))
sorting3.set_unit_spike_features(unit_id=1, feature_name='random_value',
                                 value=random_values)
print("Spike feature names: " + str(sorting3.get_unit_spike_feature_names(unit_id=1)))
```

Out:

```
Spike feature names: ['random_value']
```

Total running time of the script: (0 minutes 0.007 seconds)

6.1.3 Handling probe information

In order to properly spike sort, you may need to load information related to the probe you are using. You can easily load probe information in `spikeinterface.extractors` module.

Here's how!

```
import numpy as np
import spikeinterface.extractors as se
```

First, let's create a toy example:

```
recording, sorting_true = se.example_datasets.toy_example(duration=10, num_
↪channels=32, seed=0)
```

Probe information may be required to:

- apply a channel map
- load 'group' information
- load 'location' information
- load arbitrary information

Probe information can be loaded either using a '.prb' or a '.csv' file. We recommend using a '.prb' file, since it allows users to load several information as once.

A '.prb' file is a python dictionary. Here is the content of a sample '.prb' file (eight_tetrodes.prb), that splits the channels in 8 channel groups, applies a channel map (reversing the order of each tetrode), and loads a 'label' for each electrode (arbitrary information):

`eight_tetrodes.prb`:

```
channel_groups = {
    # Tetrode index
    0:
        {
          'channels': [3, 2, 1, 0],
          'geometry': [[0,0], [1,0], [2,0], [3,0]],
          'label': ['t_00', 't_01', 't_02', 't_03'],
        },
    1:
        {
          'channels': [7, 6, 5, 4],
          'geometry': [[6,0], [7,0], [8,0], [9,0]],
          'label': ['t_10', 't_11', 't_12', 't_13'],
        },
    2:
        {
          'channels': [11, 10, 9, 8],
          'geometry': [[12,0], [13,0], [14,0], [15,0]],
          'label': ['t_20', 't_21', 't_22', 't_23'],
        },
    3:
        {
          'channels': [15, 14, 13, 12],
          'geometry': [[18,0], [19,0], [20,0], [21,0]],
          'label': ['t_30', 't_31', 't_32', 't_33'],
         },
    4:
        {
          'channels': [19, 18, 17, 16],
          'geometry': [[30,0], [31,0], [32,0], [33,0]],
          'label': ['t_40', 't_41', 't_42', 't_43'],
        },
    5:
        {
          'channels': [23, 22, 21, 20],
          'geometry': [[36,0], [37,0], [38,0], [39,0]],
          'label': ['t_50', 't_51', 't_52', 't_53'],
        },
    6:
        {
          'channels': [27, 26, 25, 24],
          'geometry': [[42,0], [43,0], [44,0], [45,0]],
          'label': ['t_60', 't_61', 't_62', 't_63'],
        },
    7:
        {
          'channels': [31, 30, 29, 28],
          'geometry': [[48,0], [49,0], [50,0], [51,0]],
          'label': ['t_70', 't_71', 't_72', 't_73'],
        }
    }
```

You can load the probe file using the `load_probe_file` function in the RecordingExtractor. **IMPORTANT** This function returns a **new** `RecordingExtractor` object:

```
recording_tetrodes = recording.load_probe_file(probe_file='eight_tetrodes.prb')
```

Now let's check what we have loaded:

```
print('Channel ids:', recording_tetrodes.get_channel_ids())
print('Loaded properties', recording_tetrodes.get_shared_channel_property_names())
print('Label of channel 0:', recording_tetrodes.get_channel_property(channel_id=0,␣
↪property_name='label'))
```

Out:

```
Channel ids: [3, 2, 1, 0, 7, 6, 5, 4, 11, 10, 9, 8, 15, 14, 13, 12, 19, 18, 17, 16,␣
↪23, 22, 21, 20, 27, 26, 25, 24, 31, 30, 29, 28]
Loaded properties ['gain', 'group', 'label', 'location', 'offset']
Label of channel 0: t_03
```

and let's plot the probe layout:

```
import spikeinterface.widgets as sw
w_el_tetrode = sw.plot_electrode_geometry(recording_tetrodes)
```

Alternatively, one can use a '.csv' file to load the electrode locations. Let's create a '.csv' file with 2D locations in a circular layout:

```
delta_deg = 2 * np.pi / recording.get_num_channels()
with open('circular_layout.csv', 'w') as f:
    for i in range(recording.get_num_channels()):
        angle = i * delta_deg
        radius = 50
```

(continues on next page)

(continued from previous page)

```
        x = radius * np.cos(angle)
        y = radius * np.sin(angle)
        f.write(str(x) + ',' + str(y) + '\n')
```

When loading the probe file as a '.csv' file, we can also pass a 'channel_map' and a 'channel_groups' arguments. For example, let's reverse the channel order and split the channels in two groups:

```
channel_map = list(range(recording.get_num_channels()))[::-1]
channel_groups = np.array(([0] * int(recording.get_num_channels())))
channel_groups[int(recording.get_num_channels() / 2):] = 1

print('Created channel map', channel_map)
print('Created channel groups', channel_groups)
```

Out:

```
Created channel map [31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16,␣
↪15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
Created channel groups [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1␣
↪1]
```

We can now load the probe information from the newly created '.csv' file:

```
recording_circ = recording.load_probe_file(probe_file='circular_layout.csv',
                                           channel_map=channel_map,
                                           channel_groups=channel_groups)
```

Here is now the probe layout:

```
w_el_circ = sw.plot_electrode_geometry(recording_circ)
```

Let's check that we loaded the information correctly:

```
print('Loaded channel ids', recording_circ.get_channel_ids())
print('Loaded channel groups', recording_circ.get_channel_groups())
```

Out:

```
Loaded channel ids [31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16,␣
↪15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
Loaded channel groups [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1␣
↪1]
```

Total running time of the script: (0 minutes 1.100 seconds)

6.1.4 Working with unscaled traces

Some file formats store data in convenient types that require offsetting and scaling in order to convert the traces to uV. This example shows how to work with unscaled and scaled traces int `spikeinterface.extractors` module.

```
import numpy as np
import matplotlib.pyplot as plt
import spikeinterface.extractors as se
```

First, let's create some traces in unsigned int16 type. Assuming the ADC output of our recording system has 10 bits, the values will be between 0 and 1024. Let's assume our signal is centered at 512 and it has a standard deviation of 50

bits

```
sampling_frequency = 30000
traces = 512 + 50 * np.random.randn(4, 10*sampling_frequency)
traces = traces.astype("uint16")
```

Let's now instantiate a `NumpyRecordingExtractor` with the traces we just created

```
recording = se.NumpyRecordingExtractor(traces, sampling_frequency=sampling_frequency)
print(f"Traces dtype: {recording.get_dtype()}")
```

Out:

```
Traces dtype: uint16
```

Since our ADC samples between 0 and 1024, we need to convert to uV. To do so, we need to transform the traces as: traces_uV = traces_raw * gains + offset

Let's assume that our gain (i.e. the value of each bit) is 0.1, so that our voltage range is between 0 and 1024*0.1. We also need an offset to center the traces around 0. The offset will be: - 2^(10-1) * gain = -512 * gain (where 10 is the number of bits of our ADC)

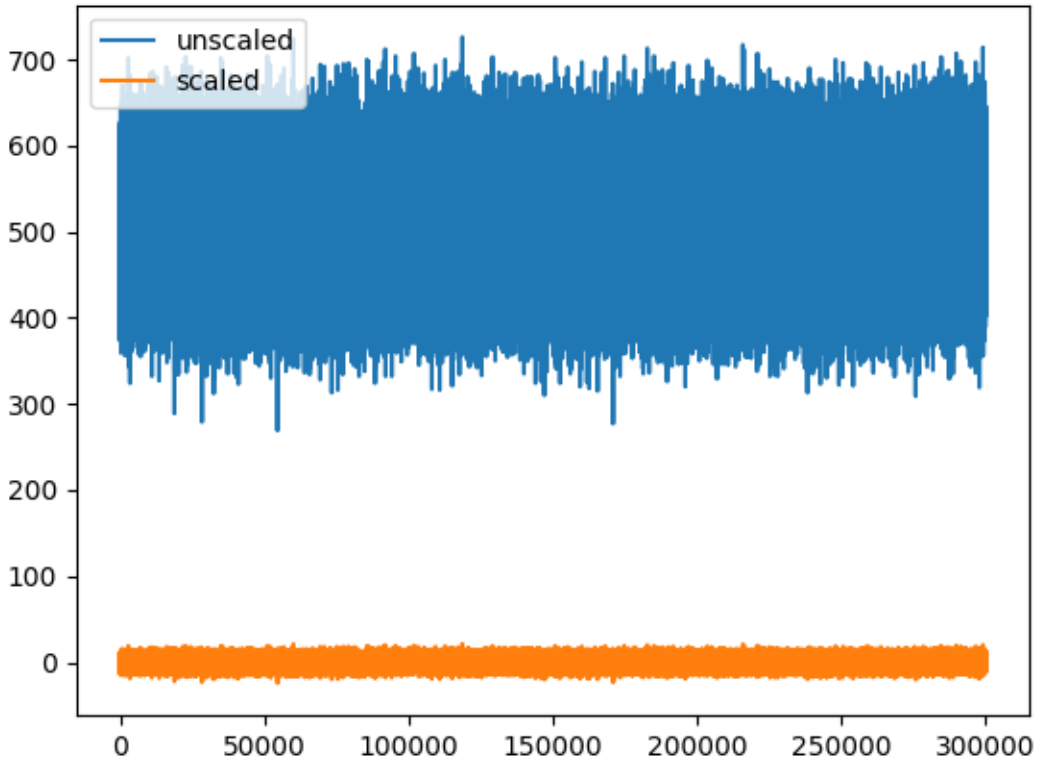
```
gain = 0.1
offset = -2**(10 - 1) * gain
```

We are now ready to set gains and offsets to our extractor. We also have to set the `has_unscaled` field to `True`:

```
recording.set_channel_gains(gain)
recording.set_channel_offsets(offset)
recording.has_unscaled = True
```

With gains and offset information, we can retrieve traces both in their unscaled (raw) type, and in their scaled type:

```
traces_unscaled = recording.get_traces(return_scaled=False)
traces_scaled = recording.get_traces(return_scaled=True)  # return_scaled is True by␣
↪default

print(f"Traces dtype after scaling: {recording.get_dtype(return_scaled=True)}")

plt.plot(traces_unscaled[0], label="unscaled")
plt.plot(traces_scaled[0], label="scaled")
plt.legend()
```



Out:

```
Traces dtype after scaling: float32

<matplotlib.legend.Legend object at 0x7f3e057a0240>
```

Total running time of the script: (0 minutes 0.846 seconds)

6.2 Toolkit tutorials

The `toolkit` module imports the spiketoolkit package. It allows users to preprocess and postprocess the data, to compute validation metrics, and to perform automatic curation of spike sorting outputs.

- preprocessing
- postprocessing
- validation
- curation

6.2.1 Preprocessing Tutorial

Before spike sorting, you may need to preproccess your signals in order to improve the spike sorting performance. You can do that in SpikeInterface using the `toolkit.preprocessing` submodule.

```
import numpy as np
import matplotlib.pylab as plt
import scipy.signal

import spikeinterface.extractors as se
import spikeinterface.toolkit as st
```

First, let's create a toy example:

```
recording, sorting = se.example_datasets.toy_example(num_channels=4, duration=10,␣
↪seed=0)
```

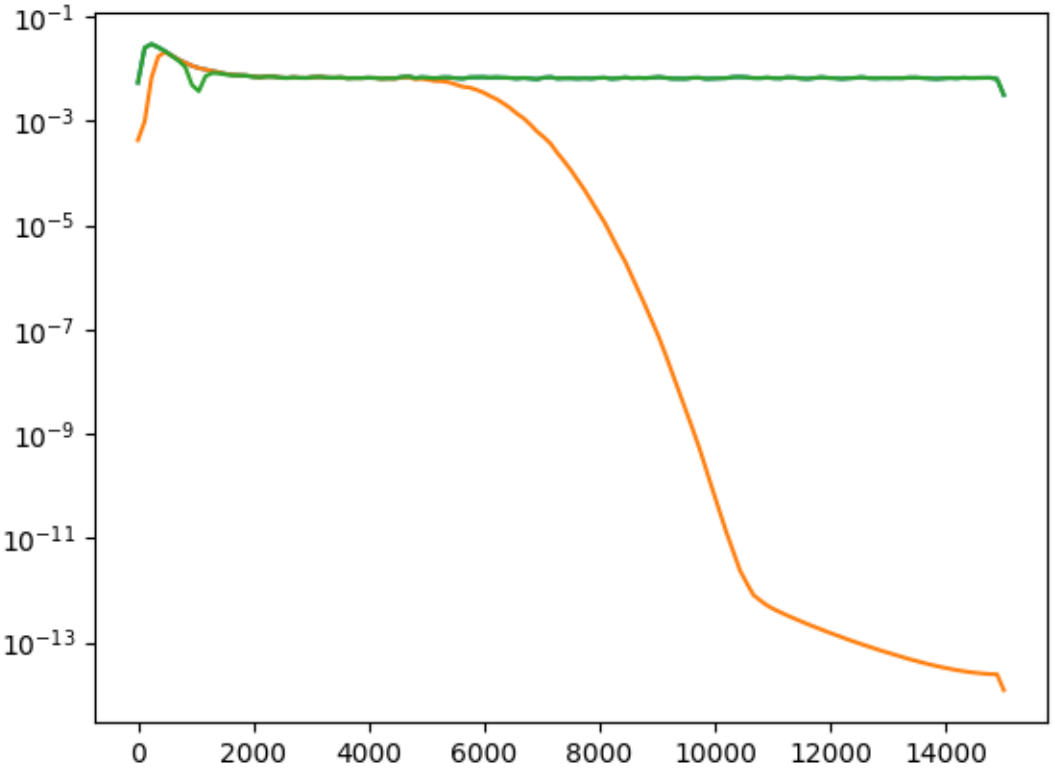
Apply filters

Now apply a bandpass filter and a notch filter (separately) to the recording extractor. Filters are also RecordingExtractor objects.

```
recording_bp = st.preprocessing.bandpass_filter(recording, freq_min=300, freq_
↪max=6000)
recording_notch = st.preprocessing.notch_filter(recording, freq=1000, q=10)
```

Now let's plot the power spectrum of non-filtered, bandpass filtered, and notch filtered recordings.

```
f_raw, p_raw = scipy.signal.welch(recording.get_traces(), fs=recording.get_sampling_
↪frequency())
f_bp, p_bp = scipy.signal.welch(recording_bp.get_traces(), fs=recording.get_sampling_
↪frequency())
f_notch, p_notch = scipy.signal.welch(recording_notch.get_traces(), fs=recording.get_
↪sampling_frequency())

fig, ax = plt.subplots()
ax.semilogy(f_raw, p_raw[0], f_bp, p_bp[0], f_notch, p_notch[0])
```



Out:

```
[<matplotlib.lines.Line2D object at 0x7f3e05894320>, <matplotlib.lines.Line2D object␣
↪at 0x7f3e0589f080>, <matplotlib.lines.Line2D object at 0x7f3e0589f588>]
```

Compute LFP and MUA

Local field potentials (LFP) are low frequency components of the extracellular recordings. Multi-unit activity (MUA) are rectified and low-pass filtered recordings showing the diffuse spiking activity.

In `spiketoolkit`, LFP and MUA can be extracted combining the `bandpass_filter`, `rectify` and `resample` functions. In this example LFP and MUA are resampled at 1000 Hz.

```
recording_lfp = st.preprocessing.bandpass_filter(recording, freq_min=1, freq_max=300)
recording_lfp = st.preprocessing.resample(recording_lfp, 1000)
recording_mua = st.preprocessing.resample(st.preprocessing.rectify(recording), 1000)
```

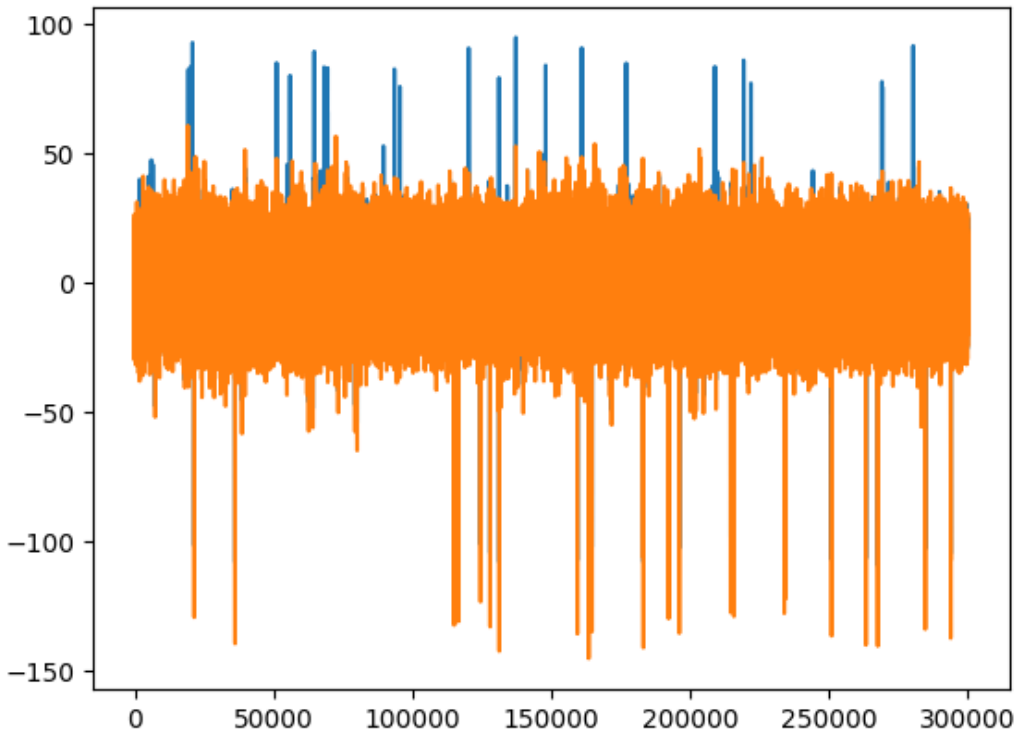
The toy example data are only contain high frequency components, but these lines of code will work on experimental data

Change reference

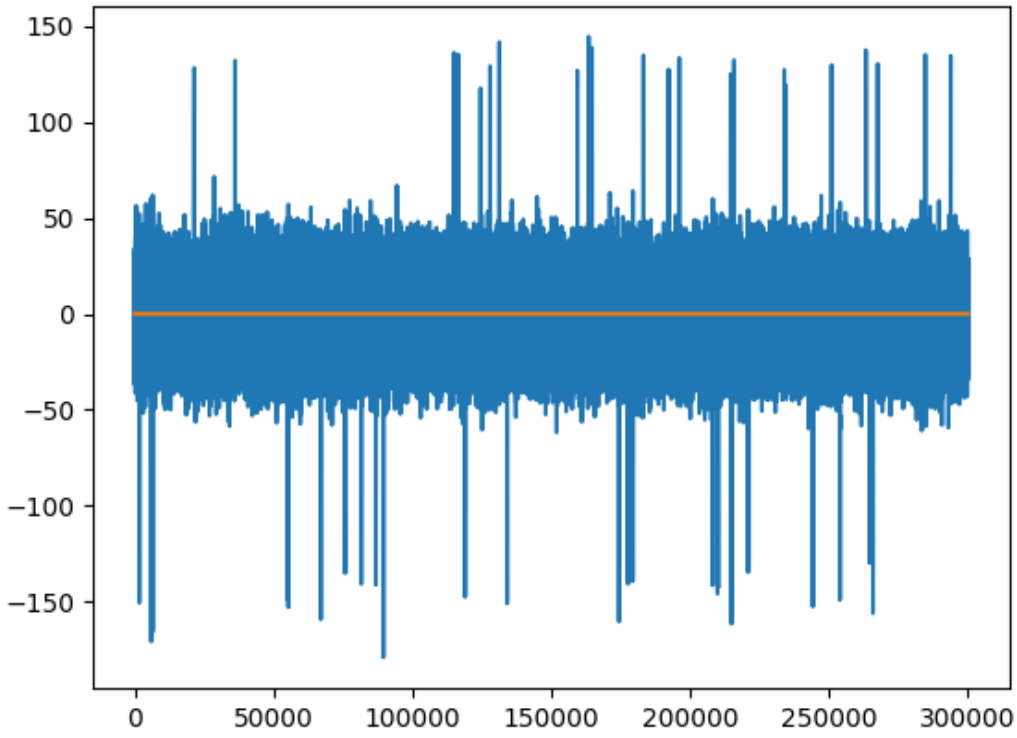
In many cases, before spike sorting, it is wise to re-reference the signals to reduce the common-mode noise from the recordings.

To re-reference in `spiketoolkit` you can use the `common_reference` function. Both common average reference (CAR) and common median reference (CMR) can be applied. Moreover, the average/median can be computed on different groups. Single channels can also be used as reference.

```
recording_car = st.preprocessing.common_reference(recording, reference='average')
recording_cmr = st.preprocessing.common_reference(recording, reference='median')
recording_single = st.preprocessing.common_reference(recording, reference='single',␣
↪ref_channels=[0])
recording_single_groups = st.preprocessing.common_reference(recording, reference=
↪'single',
                                                            groups=[[0, 1], [2, 3]],␣
↪ref_channels=[0, 2])


fig1, ax1 = plt.subplots()
ax1.plot(recording_car.get_traces()[0])
ax1.plot(recording_cmr.get_traces()[0])

fig2, ax2 = plt.subplots()
ax2.plot(recording_single_groups.get_traces()[1])  # not zero
ax2.plot(recording_single_groups.get_traces()[0])
```



•



Out:

```
[<matplotlib.lines.Line2D object at 0x7f3e056fdb38>]
```

Remove bad channels

In to remove noisy channels from the analysis, the `remove_bad_channels` function can be used.

```
recording_remove_bad = st.preprocessing.remove_bad_channels(recording, bad_channel_
↪ids=[0])

print(recording_remove_bad.get_channel_ids())
```

Out:

```
[1, 2, 3]
```

As expected, channel 0 is removed. Bad channels removal can also be done automatically. In this case, the channels with a standard deviation exceeding `bad_threshold` times the median standard deviation are removed. The standard deviations are computed on the traces with length `seconds` from the middle of the recordings.

```
recording_remove_bad_auto = st.preprocessing.remove_bad_channels(recording, bad_
↪channel_ids=None, bad_threshold=2,
                                                                 seconds=2)

print(recording_remove_bad_auto.get_channel_ids())
```

Out:

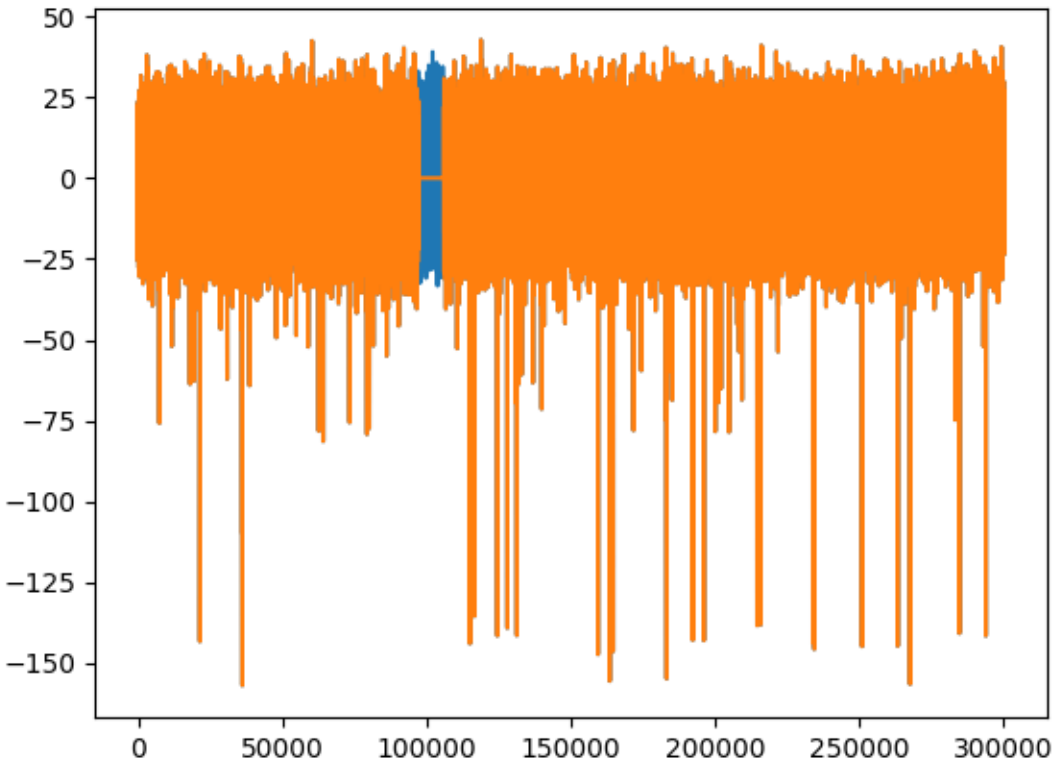
```
[0, 1, 2, 3]
```

With these simulated recordings, there are no noisy channel.

Remove stimulation artifacts

In some applications, electrodes are used to electrically stimulate the tissue, generating a large artifact. In `spiketoolkit`, the artifact can be zeroed-out using the `remove_artifact` function.

```
# create dummy stimulation triggers
stimulation_trigger_frames = np.array([100000, 500000, 700000])

# large ms_before and s_after are used for plotting only
recording_rmartifact = st.preprocessing.remove_artifacts(recording,
                                                         triggers=stimulation_trigger_
↪frames,
                                                         ms_before=100, ms_after=200)

fig3, ax3 = plt.subplots()
ax3.plot(recording.get_traces()[0])
ax3.plot(recording_rmartifact.get_traces()[0])
```



Out:

```
[<matplotlib.lines.Line2D object at 0x7f3e05684320>]
```

You can list the available preprocessors with:

```
print(st.preprocessing.preprocessers_full_list)
```

Out:

```
[<class 'spiketoolkit.preprocessing.highpass_filter.HighpassFilterRecording'>, <class
↪'spiketoolkit.preprocessing.bandpass_filter.BandpassFilterRecording'>, <class
↪'spiketoolkit.preprocessing.notch_filter.NotchFilterRecording'>, <class
↪'spiketoolkit.preprocessing.whiten.WhitenRecording'>, <class 'spiketoolkit.
↪preprocessing.common_reference.CommonReferenceRecording'>, <class 'spiketoolkit.
↪preprocessing.resample.ResampleRecording'>, <class 'spiketoolkit.preprocessing.
↪rectify.RectifyRecording'>, <class 'spiketoolkit.preprocessing.remove_artifacts.
↪RemoveArtifactsRecording'>, <class 'spiketoolkit.preprocessing.remove_bad_channels.
↪RemoveBadChannelsRecording'>, <class 'spiketoolkit.preprocessing.transform.
↪TransformRecording'>, <class 'spiketoolkit.preprocessing.normalize_by_quantile.
↪NormalizeByQuantileRecording'>, <class 'spiketoolkit.preprocessing.clip.
↪ClipRecording'>, <class 'spiketoolkit.preprocessing.blank_saturation.
↪BlankSaturationRecording'>, <class 'spiketoolkit.preprocessing.center.
↪CenterRecording'>, <class 'spiketoolkit.preprocessing.mask.MaskRecording'>]
```

Total running time of the script: (0 minutes 2.023 seconds)

6.2.2 Postprocessing Tutorial

Spike sorters generally output a set of units with corresponding spike trains. The `toolkit.postprocessing` submodule allows to combine the `RecordingExtractor` and the sorted `SortingExtractor` objects to perform further postprocessing.

```
import matplotlib.pylab as plt

import spikeinterface.extractors as se
import spikeinterface.toolkit as st
```

First, let's create a toy example:

```
recording, sorting = se.example_datasets.toy_example(num_channels=4, duration=10,␣
↪seed=0)
```

Assuming the `sorting` is the output of a spike sorter, the `postprocessing` module allows to extract all relevant information from the paired recording-sorting.

Compute spike waveforms

Waveforms are extracted with the `get_unit_waveforms` function by extracting snippets of the recordings when spikes are detected. When waveforms are extracted, the can be loaded in the `SortingExtractor` object as features. The ms before and after the spike event can be chosen. Waveforms are returned as a list of np.arrays (n_spikes, n_channels, n_points)

```
wf = st.postprocessing.get_unit_waveforms(recording, sorting, ms_before=1, ms_after=2,
                                          save_as_features=True, verbose=True)
```

Out:

```
Number of chunks: 1 - Number of jobs: 1

Extracting waveforms in chunks:   0%|          | 0/1 [00:00<?, ?it/s]
Extracting waveforms in chunks: 100%|##########| 1/1 [00:00<00:00, 272.32it/s]
```

Now `waveforms` is a unit spike feature!

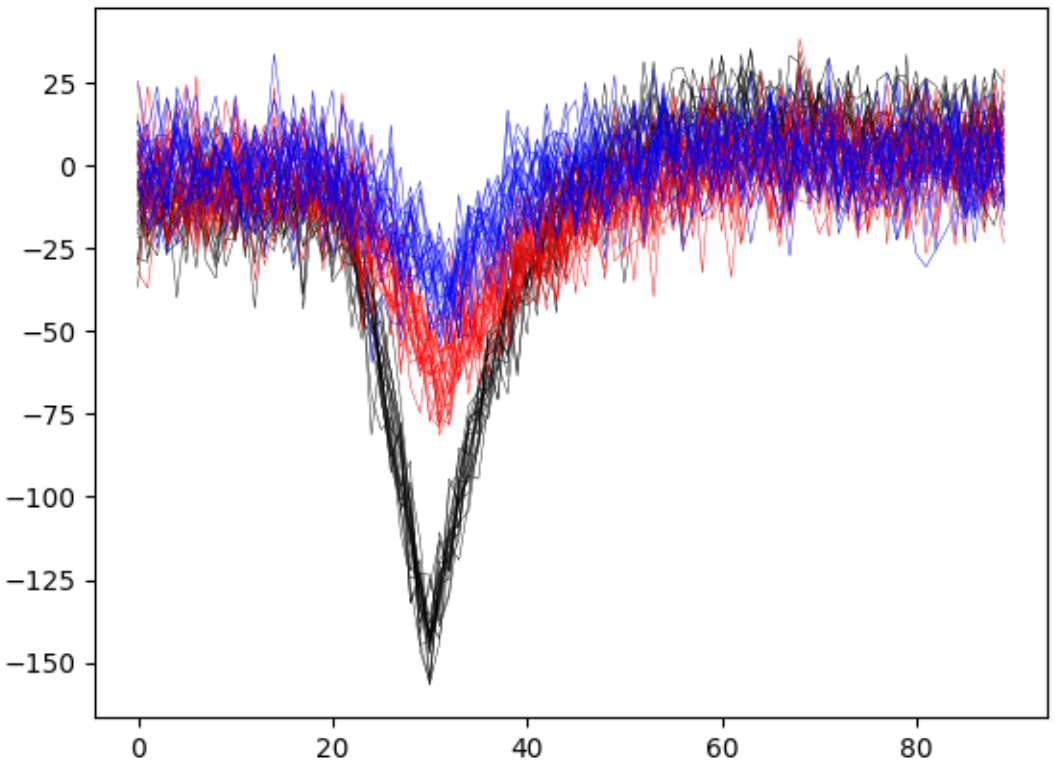
```
print(sorting.get_shared_unit_spike_feature_names())
print(wf[0].shape)
```

Out:

```
['waveforms', 'waveforms_idxs']
(22, 4, 90)
```

plotting waveforms of units 0,1,2 on channel 0

```
fig, ax = plt.subplots()
ax.plot(wf[0][:, 0, :].T, color='k', lw=0.3)
ax.plot(wf[1][:, 0, :].T, color='r', lw=0.3)
ax.plot(wf[2][:, 0, :].T, color='b', lw=0.3)
```



Out:

```
[<matplotlib.lines.Line2D object at 0x7f3e05737550>, <matplotlib.lines.Line2D object␣
↪at 0x7f3e057379b0>, <matplotlib.lines.Line2D object at 0x7f3e05737c50>, <matplotlib.
↪lines.Line2D object at 0x7f3e05737278>, <matplotlib.lines.Line2D object at␣
↪0x7f3e057370f0>, <matplotlib.lines.Line2D object at 0x7f3e05737748>, <matplotlib.
↪lines.Line2D object at 0x7f3e05737fd0>, <matplotlib.lines.Line2D object at␣
↪0x7f3e05737978>, <matplotlib.lines.Line2D object at 0x7f3e05737400>, <matplotlib.
↪lines.Line2D object at 0x7f3e059fa5c0>, <matplotlib.lines.Line2D object at␣
↪0x7f3e059fa240>, <matplotlib.lines.Line2D object at 0x7f3e059fab70>, <matplotlib.
↪lines.Line2D object at 0x7f3e05886f60>, <matplotlib.lines.Line2D object at␣
```

(continues on next page)

(continued from previous page)

If the a certain property (e.g. `group`) is present in the RecordingExtractor, the waveforms can be extracted only on the channels with that property using the `grouping_property` and `compute_property_from_recording` arguments. For example, if channel [0,1] are in group 0 and channel [2,3] are in group 2, then if the peak of the waveforms is in channel [0,1] it will be assigned to group 0 and will have 2 channels and the same for group 1.

```
channel_groups = [[0, 1], [2, 3]]
for ch in recording.get_channel_ids():
    for gr, channel_group in enumerate(channel_groups):
        if ch in channel_group:
            recording.set_channel_property(ch, 'group', gr)
print(recording.get_channel_property(0, 'group'), recording.get_channel_property(2,
↪'group'))
```

Out:

```
0 1
```

```
wf_by_group = st.postprocessing.get_unit_waveforms(recording, sorting, ms_before=1,␣
↪ms_after=2,
                                                   save_as_features=False,␣
↪verbose=True,
                                                   grouping_property='group',
                                                   compute_property_from_
↪recording=True)

# now waveforms will only have 2 channels
print(wf_by_group[0].shape)
```

Out:

```
(22, 4, 90)
```

Compute unit templates

Similarly to waveforms, templates - average waveforms - can be easily extracted using the `get_unit_templates`. When spike trains have numerous spikes, you can set the `max_spikes_per_unit` to be extracted. If waveforms have already been computed and stored as `features`, those will be used. Templates can be saved as unit properties.

```
templates = st.postprocessing.get_unit_templates(recording, sorting, max_spikes_per_
↪unit=200,
                                                 save_as_property=True, verbose=True)
```

```
print(sorting.get_shared_unit_property_names())
```

Out:

```
['template']
```

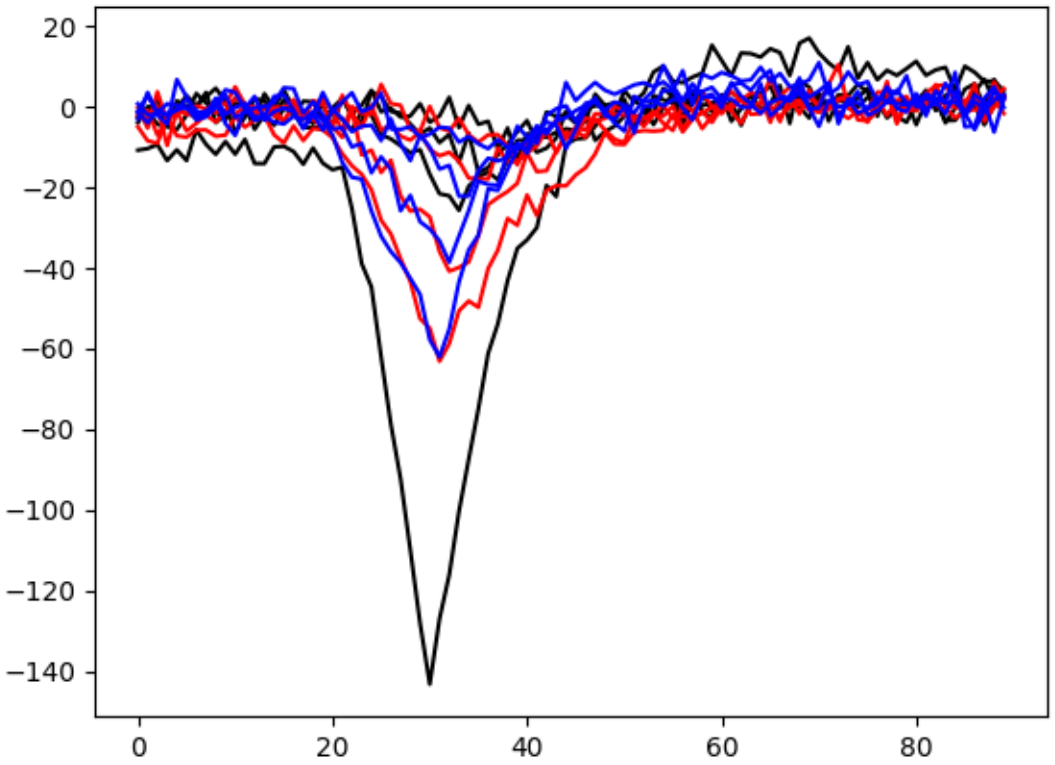
Plotting templates of units 0,1,2 on all four channels

```
fig, ax = plt.subplots()
ax.plot(templates[0].T, color='k')
```

(continues on next page)

(continued from previous page)

```
ax.plot(templates[1].T, color='r')
ax.plot(templates[2].T, color='b')
```



Out:

```
[<matplotlib.lines.Line2D object at 0x7f3e057ac5c0>, <matplotlib.lines.Line2D object␣
↪at 0x7f3e057ac710>, <matplotlib.lines.Line2D object at 0x7f3e057ac860>, <matplotlib.
↪lines.Line2D object at 0x7f3e057ac9b0>]
```

Compute unit maximum channel ————————————-

In the same way, one can get the ecording channel with the maximum amplitude and save it as a property.

```
max_chan = st.postprocessing.get_unit_max_channels(recording, sorting, save_as_
↪property=True, verbose=True)
print(max_chan)
```

Out:

```
[0, 0, 1, 1, 1, 2, 2, 2, 3, 3]
```

```
print(sorting.get_shared_unit_property_names())
```

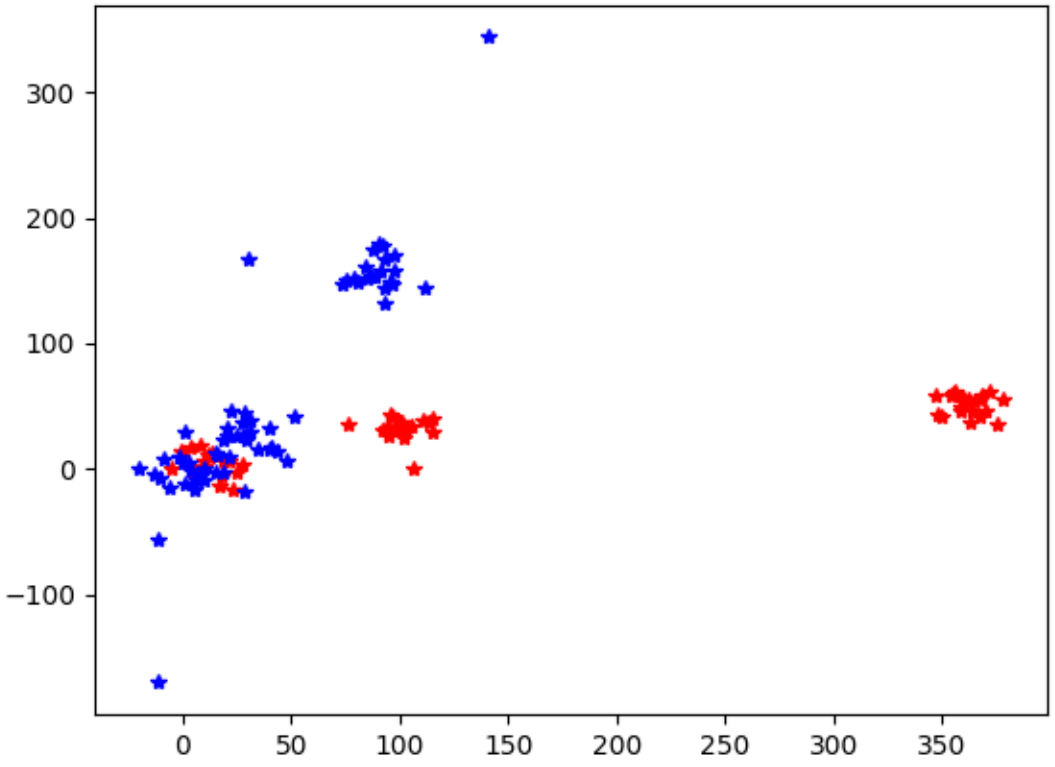
Out:

```
['max_channel', 'template']
```

Compute pca scores ——————————

For some applications, for example validating the spike sorting output, PCA scores can be computed.

```
pca_scores = st.postprocessing.compute_unit_pca_scores(recording, sorting, n_comp=3,␣
↪verbose=True)

for pc in pca_scores:
    print(pc.shape)

fig, ax = plt.subplots()
ax.plot(pca_scores[0][:, 0], pca_scores[0][:, 1], 'r*')
ax.plot(pca_scores[2][:, 0], pca_scores[2][:, 1], 'b*')
```



Out:

```
Computing waveforms
Fitting PCA of 3 dimensions on 241 waveforms
Projecting waveforms on PC
(22, 4, 3)
(26, 4, 3)
(22, 4, 3)
(25, 4, 3)
```

(continues on next page)

(continued from previous page)

```
(25, 4, 3)
(27, 4, 3)
(22, 4, 3)
(22, 4, 3)
(28, 4, 3)
(22, 4, 3)

[<matplotlib.lines.Line2D object at 0x7f3e056ddda0>, <matplotlib.lines.Line2D object␣
↪at 0x7f3e056dd748>, <matplotlib.lines.Line2D object at 0x7f3e056dd9b0>]
```

PCA scores can be also computed electrode-wise. In the previous example, PCA was applied to the concatenation of the waveforms over channels.

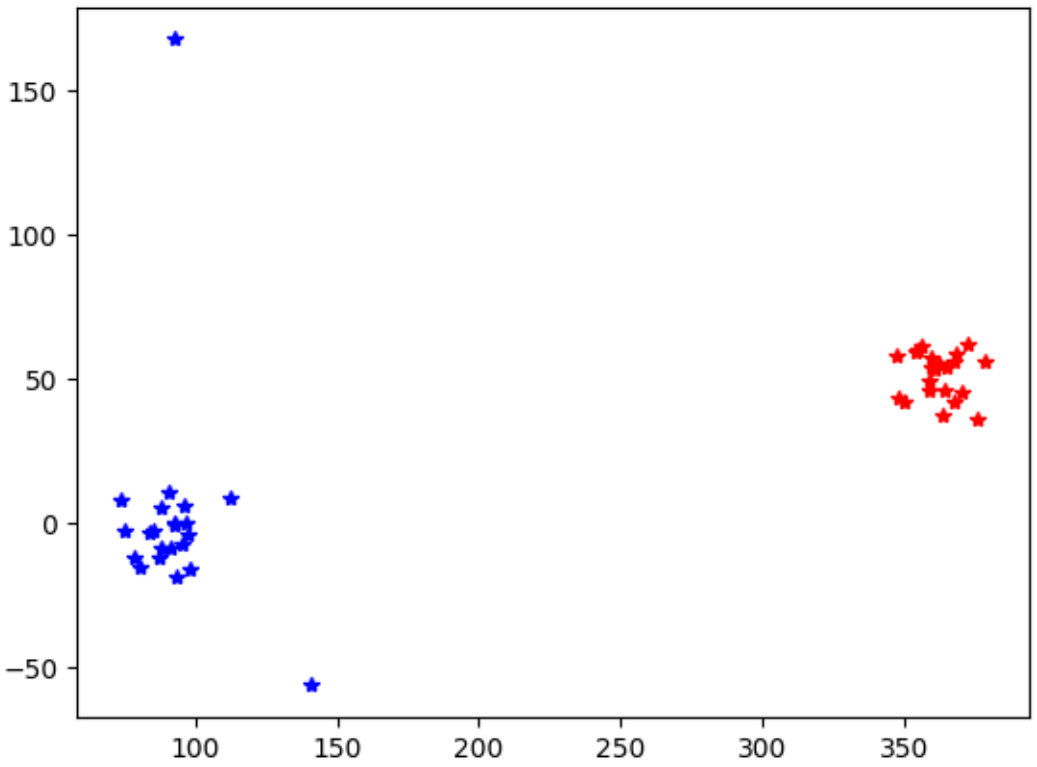
```
pca_scores_by_electrode = st.postprocessing.compute_unit_pca_scores(recording,␣
↪sorting, n_comp=3, by_electrode=True)

for pc in pca_scores_by_electrode:
    print(pc.shape)
```

Out:

```
(22, 4, 3)
(26, 4, 3)
(22, 4, 3)
(25, 4, 3)
(25, 4, 3)
(27, 4, 3)
(22, 4, 3)
(22, 4, 3)
(28, 4, 3)
(22, 4, 3)
```

In this case, as expected, 3 principal components are extracted for each electrode.

```
fig, ax = plt.subplots()
ax.plot(pca_scores_by_electrode[0][:, 0, 0], pca_scores_by_electrode[0][:, 1, 0], 'r*
↪')
ax.plot(pca_scores_by_electrode[2][:, 0, 0], pca_scores_by_electrode[2][:, 1, 1], 'b*
↪')
```



Out:

```
[<matplotlib.lines.Line2D object at 0x7f3e04498e48>]
```

Export sorted data to Phy for manual curation

Finally, it is common to visualize and manually curate the data after spike sorting. In order to do so, we interface wiht the Phy (https://phy-contrib.readthedocs.io/en/latest/template-gui/).

First, we need to export the data to the phy format:

```
st.postprocessing.export_to_phy(recording, sorting, output_folder='phy', verbose=True)
```

Out:

```
Converting to Phy format
Saving files
Saved phy format to:  /home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/
↪checkouts/0.13.0/examples/modules/toolkit/phy
Run:

phy template-gui  /home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/
↪checkouts/0.13.0/examples/modules/toolkit/phy/params.py
```

To run phy you can then run (from terminal): `phy template-gui phy/params.py`

Or from a notebook: `!phy template-gui phy/params.py`

After manual curation you can load back the curated data using the `PhySortingExtractor`:

Total running time of the script: (0 minutes 0.677 seconds)

6.2.3 Validation Tutorial

After spike sorting, you might want to validate the goodness of the sorted units. This can be done using the `toolkit.validation` submodule, which computes several quality metrics of the sorted units.

```
import spikeinterface.extractors as se
import spikeinterface.toolkit as st
```

First, let's create a toy example:

```
recording, sorting = se.example_datasets.toy_example(num_channels=4, duration=10,␣
↪seed=0)
```

The `toolkit.validation` submodule has a set of functions that allow users to compute metrics in a compact and easy way. To compute a single metric, the user can simply run one of the quality metric functions as shown below. Each function as a variety of adjustable parameters that can be tuned by the user to match their data.

```
firing_rates = st.validation.compute_firing_rates(sorting, duration_in_
↪frames=recording.get_num_frames())
isi_violations = st.validation.compute_isi_violations(sorting, duration_in_
↪frames=recording.get_num_frames(), isi_threshold=0.0015)
snrs = st.validation.compute_snrs(recording=recording, sorting=sorting, max_spikes_
↪per_unit_for_snr=1000)
nn_hit_rate, nn_miss_rate = st.validation.compute_nn_metrics(recording=recording,␣
↪sorting=sorting, num_channels_to_compare=13)
```

To compute more than one metric at once, a user can use the `compute_quality_metrics` function and indicate which metrics they want to compute. This will return a dictionary of metrics or optionally a pandas dataframe.

```
metrics = st.validation.compute_quality_metrics(sorting=sorting, recording=recording,
                                                metric_names=['firing_rate', 'isi_
↪violation', 'snr', 'nn_hit_rate', 'nn_miss_rate'],
                                                as_dataframe=True)
```

To compute metrics on only part of the recording, a user can specify specific epochs in the Recording and Sorting extractor using `add_epoch` and then compute the metrics on the SubRecording and SubSorting extractor given by `get_epoch`. In this example, we compute all the same metrics on the first half of the recording.

```
sorting.add_epoch(epoch_name="first_half", start_frame=0, end_frame=recording.get_num_
↪frames()/2) #set
recording.add_epoch(epoch_name="first_half", start_frame=0, end_frame=recording.get_
↪num_frames()/2)
subsorting = sorting.get_epoch("first_half")
subrecording = recording.get_epoch("first_half")
metrics_first_half = st.validation.compute_quality_metrics(sorting=subsorting,␣
↪recording=subrecording,
                                                           metric_names=['firing_rate
↪', 'isi_violation', 'snr', 'nn_hit_rate', 'nn_miss_rate'],
                                                           as_dataframe=True)

print("Metrics full recording")
```

(continues on next page)

(continued from previous page)

```
print(metrics)
print('\n')
print("Metrics first half recording")
print(metrics_first_half)
```

Out:

```
Metrics full recording
    firing_rate  isi_violation        snr  nn_hit_rate  nn_miss_rate
1           2.2            0.0  14.190699     1.000000      0.000000
2           2.6            0.0   6.252029     0.974359      0.004651
3           2.2            0.0   6.107215     0.878788      0.007610
4           2.5            0.0  15.809131     1.000000      0.004630
5           2.5            0.0   3.781591     0.893333      0.018519
6           2.7            0.0   3.945409     0.938272      0.009346
7           2.2            0.0  28.892361     1.000000      0.000000
8           2.2            0.0   7.082476     0.954545      0.000000
9           2.8            0.0   8.048143     0.964286      0.003130
10          2.2            0.0   4.093077     0.893939      0.007610


Metrics first half recording
    firing_rate  isi_violation        snr  nn_hit_rate  nn_miss_rate
1           1.4            0.0  14.225538     1.000000      0.000000
2           3.2            0.0   6.267378     0.979167      0.011494
3           3.2            0.0   6.092624     0.895833      0.011494
4           2.4            0.0  15.771360     1.000000      0.000000
5           2.6            0.0   3.772556     0.820513      0.019608
6           3.6            0.0   3.932851     0.907407      0.029240
7           2.8            0.0  28.800394     1.000000      0.000000
8           2.2            0.0   7.059932     0.909091      0.000000
9           2.8            0.0   8.035229     0.976190      0.002825
10          2.2            0.0   4.086510     0.787879      0.008264
```

Total running time of the script: (0 minutes 0.480 seconds)

6.2.4 Curation Tutorial

After spike sorting and computing validation metrics, you can automatically curate the spike sorting output using the quality metrics. This can be done with the `toolkit.curation` submodule.

```
import spikeinterface.extractors as se
import spikeinterface.toolkit as st
import spikeinterface.sorters as ss
```

First, let's create a toy example:

```
recording, sorting = se.example_datasets.toy_example(num_channels=4, duration=30,␣
↪seed=0)
```

and let's spike sort using klusta

```
sorting_KL = ss.run_klusta(recording)

print('Units:', sorting_KL.get_unit_ids())
print('Number of units:', len(sorting_KL.get_unit_ids()))
```

Out:

```
RUNNING SHELL SCRIPT: /home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/
↪checkouts/0.13.0/examples/modules/toolkit/klusta_output/run_klusta.sh
/home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/checkouts/0.13.0/doc/
↪sources/spikesorters/spikesorters/basesorter.py:158: ResourceWarning: unclosed file
↪<_io.TextIOWrapper name=63 encoding='UTF-8'>
  self._run(recording, self.output_folders[i])
Units: [0, 2, 3, 4, 5, 6, 7, 8, 9]
Number of units: 9
```

There are several available functions that enables to only retrieve units with respect to some rules. For example, let's automatically curate the sorting output so that only the units with SNR > 10 and mean firing rate > 2.3 Hz are kept:

```
sorting_fr = st.curation.threshold_firing_rates(sorting_KL, duration_in_
↪frames=recording.get_num_frames(), threshold=2.3, threshold_sign='less')

print('Units after FR theshold:', sorting_fr.get_unit_ids())
print('Number of units after FR theshold:', len(sorting_fr.get_unit_ids()))

sorting_snr = st.curation.threshold_snrs(sorting_fr, recording, threshold=10,␣
↪threshold_sign='less')

print('Units after SNR theshold:', sorting_snr.get_unit_ids())
print('Number of units after SNR theshold:', len(sorting_snr.get_unit_ids()))
```

Out:

```
Units after FR theshold: [0, 3]
Number of units after FR theshold: 2
Units after SNR theshold: [3]
Number of units after SNR theshold: 1
```

Let's now check with the `toolkit.validation` submodule that all units have a firing rate > 10 and snr > 0

```
fr = st.validation.compute_firing_rates(sorting_snr, duration_in_frames=recording.get_
↪num_frames())
snrs = st.validation.compute_snrs(sorting_snr, recording)

print('Firing rates:', fr)
print('SNR:', snrs)
```

Out:

```
Firing rates: [2.33333333]
SNR: [14.61125006]
```

Total running time of the script: (0 minutes 4.415 seconds)

6.3 Sorters tutorials

The `sorters` module imports the spikesorters package. It wraps several spike sorting algorithms with the same simple Python API.

- run sorters with different parameters
- spike sort by property

- use the sorter launcher

6.3.1 Run spike sorting algorithms

This example shows the basic usage of the `sorters` module of `spikeinterface`

```
import spikeinterface.extractors as se
import spikeinterface.sorters as ss
```

First, let's create a toy example:

```
recording, sorting_true = se.example_datasets.toy_example(duration=10, seed=0)
```

Check available sorters

```
print(ss.available_sorters())
```

Out:

```
['combinato', 'hdsort', 'herdingspikes', 'ironclust', 'kilosort', 'kilosort2',
↪'kilosort2_5', 'kilosort3', 'klusta', 'mountainsort4', 'spykingcircus', 'tridesclous
↪', 'waveclus', 'yass']
```

This will list the sorters available through SpikeInterface. To see which sorters are installed on the machine you can run:

```
print(ss.installed_sorters())
```

Out:

```
['klusta', 'mountainsort4', 'tridesclous']
```

Change sorter parameters

```
default_ms4_params = ss.Mountainsort4Sorter.default_params()
print(default_ms4_params)
```

Out:

```
{'detect_sign': -1, 'adjacency_radius': -1, 'freq_min': 300, 'freq_max': 6000, 'filter
↪': True, 'whiten': True, 'curation': False, 'num_workers': None, 'clip_size': 50,
↪'detect_threshold': 3, 'detect_interval': 10, 'noise_overlap_threshold': 0.15}
```

Parameters can be changed either by passing a full dictionary, or by passing single arguments.

```
# Mountainsort4 spike sorting
default_ms4_params['detect_threshold'] = 4
default_ms4_params['curation'] = False

# parameters set by params dictionary
sorting_MS4 = ss.run_mountainsort4(recording=recording, **default_ms4_params,
                                   output_folder='tmp_MS4')
```

Out:

```
Warning! The recording is already filtered, but Mountainsort4 filter is enabled. You␣
↪can disable filters by setting 'filter' parameter to False
```

```
# parameters set by params dictionary
sorting_MS4_10 = ss.run_mountainsort4(recording=recording, detect_threshold=10,
                                      output_folder='tmp_MS4')
```

Out:

```
Warning! The recording is already filtered, but Mountainsort4 filter is enabled. You␣
↪can disable filters by setting 'filter' parameter to False
```

```
print('Units found with threshold = 4:', sorting_MS4.get_unit_ids())
print('Units found with threshold = 10:', sorting_MS4_10.get_unit_ids())
```

Out:

```
Units found with threshold = 4: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Units found with threshold = 10: [1, 2, 3, 4]
```

Run other sorters

```
# SpyKING Circus spike sorting
# sorting_SC = ss.run_spykingcircus(recording, output_folder='tmp_SC')
# print('Units found with Spyking Circus:', sorting_SC.get_unit_ids())
```

```
# KiloSort spike sorting (KILOSORT_PATH and NPY_MATLAB_PATH can be set as environment␣
↪variables)
# sorting_KS = ss.run_kilosort(recording, output_folder='tmp_KS')
#  print('Units found with Kilosort:', sorting_KS.get_unit_ids())
```

```
# Kilosort2 spike sorting (KILOSORT2_PATH and NPY_MATLAB_PATH can be set as␣
↪environment variables)
# sorting_KS2 = ss.run_kilosort2(recording, output_folder='tmp_KS2')
#  print('Units found with Kilosort2', sorting_KS2.get_unit_ids())
```

```
# Klusta spike sorting
#  sorting_KL = ss.run_klusta(recording, output_folder='tmp_KL')
# print('Units found with Klusta:', sorting_KL.get_unit_ids())
```

```
# IronClust spike sorting (IRONCLUST_PATH can be set as environment variables)
# sorting_IC = ss.run_ironclust(recording, output_folder='tmp_IC')
# print('Units found with Ironclust:', sorting_IC.get_unit_ids())
```

```
# Tridesclous spike sorting
# sorting_TDC = ss.run_tridesclous(recording, output_folder='tmp_TDC')
# print('Units found with Tridesclous:', sorting_TDC.get_unit_ids())
```

Total running time of the script: (0 minutes 2.342 seconds)

6.3.2 Use the spike sorting launcher

This example shows how to use the spike sorting launcher. The launcher allows to parameterize the sorter name and to run several sorters on one or multiple recordings.

```
import spikeinterface.extractors as se
import spikeinterface.sorters as ss
```

First, let's create the usueal toy example:

```
recording, sorting_true = se.example_datasets.toy_example(duration=10, seed=0)
```

The launcher enables to call any spike sorter with the same functions: `run_sorter` and `run_sorters`. For running multiple sorters on the same recording extractor or a collection of them, the `run_sorters` function can be used.

Let's first see how to run a single sorter, for example, Klusta:

```
# The sorter name can be now a parameter, e.g. chosen with a command line interface␣
↪or a GUI
sorter_name = 'klusta'
sorting_KL = ss.run_sorter(sorter_name_or_class='klusta', recording=recording, output_
↪folder='my_sorter_output')
print(sorting_KL.get_unit_ids())
```

Out:

```
RUNNING SHELL SCRIPT: /home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/
↪checkouts/0.13.0/examples/modules/sorters/my_sorter_output/run_klusta.sh
/home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/checkouts/0.13.0/doc/
↪sources/spikesorters/spikesorters/basesorter.py:158: ResourceWarning: unclosed file
↪<_io.TextIOWrapper name=63 encoding='UTF-8'>
  self._run(recording, self.output_folders[i])
[0, 2, 3, 4, 5, 6]
```

This will launch the klusta sorter on the recording object.

You can also run multiple sorters on the same recording:

```
recording_list = [recording]
sorter_list = ['klusta', 'mountainsort4', 'tridesclous']
sorting_output = ss.run_sorters(sorter_list, recording_list, working_folder='tmp_some_
↪sorters', mode='overwrite')
```

Out:

```
RUNNING SHELL SCRIPT: /home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/
↪checkouts/0.13.0/examples/modules/sorters/tmp_some_sorters/recording_0/klusta/run_
↪klusta.sh
/home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/checkouts/0.13.0/doc/
↪sources/spikesorters/spikesorters/basesorter.py:158: ResourceWarning: unclosed file
↪<_io.TextIOWrapper name=63 encoding='UTF-8'>
  self._run(recording, self.output_folders[i])
Warning! The recording is already filtered, but Mountainsort4 filter is enabled. You␣
↪can disable filters by setting 'filter' parameter to False
{'chunksize': 3000,
 'clean_cluster': {'apply_auto_merge_cluster': True,
                   'apply_auto_split': True,
```

(continues on next page)

(continued from previous page)

```
                  'apply_trash_low_extremum': True,
                  'apply_trash_not_aligned': True,
                  'apply_trash_small_cluster': True},
 'clean_peaks': {'alien_value_threshold': None, 'mode': 'extremum_amplitude'},
 'cluster_kargs': {'adjacency_radius_um': 0.0,
                   'high_adjacency_radius_um': 0.0,
                   'max_loop': 1000,
                   'min_cluster_size': 20},
 'cluster_method': 'pruningshears',
 'duration': 10.0,
 'extract_waveforms': {'wf_left_ms': -2.0, 'wf_right_ms': 3.0},
 'feature_kargs': {'n_components': 8},
 'feature_method': 'global_pca',
 'make_catalogue': {'inter_sample_oversampling': False,
                    'sparse_thresh_level2': 3,
                    'subsample_ratio': 'auto'},
 'memory_mode': 'memmap',
 'mode': 'dense',
 'n_jobs': -1,
 'n_spike_for_centroid': 350,
 'noise_snippet': {'nb_snippet': 300},
 'peak_detector': {'adjacency_radius_um': 200.0,
                   'engine': 'numpy',
                   'method': 'global',
                   'peak_sign': '-',
                   'peak_span_ms': 0.7,
                   'relative_threshold': 5,
                   'smooth_radius_um': None},
 'peak_sampler': {'mode': 'rand', 'nb_max': 20000, 'nb_max_by_channel': None},
 'preprocessor': {'common_ref_removal': False,
                  'engine': 'numpy',
                  'highpass_freq': 400.0,
                  'lostfront_chunksize': -1,
                  'lowpass_freq': 5000.0,
                  'smooth_size': 0},
 'sparse_threshold': 1.5}
```

The 'mode' argument allows to 'overwrite' the 'working_folder' (if existing), 'raise' and Exception, or 'keep' the folder and skip the spike sorting run.

To 'sorting_output' is a dictionary that has (recording, sorter) pairs as keys and the correspondent `SortingExtractor` as values. It can be accessed as follows:

```
for (rec, sorter), sort in sorting_output.items():
    print(rec, sorter, ':', sort.get_unit_ids())
```

Out:

```
recording_0 tridesclous : [0, 1, 2, 3, 4]
recording_0 mountainsort4 : [1, 2, 3, 4, 5, 6, 7, 8, 9]
recording_0 klusta : [0, 2, 3, 4, 5, 6]
```

With the same mechanism, you can run several spike sorters on many recordings, just by creating a list of `RecordingExtractor` objects (`recording_list`).

Total running time of the script: (0 minutes 8.634 seconds)

6.3.3 Run spike sorting by property

Sometimes you may want to spike sort different electrodes separately. For example your probe can have several channel groups (for example tetrodes) or you might want to spike sort different brain regions separately, In these cases, you can spike sort by property.

```
import spikeinterface.extractors as se
import spikeinterface.sorters as ss
import time
```

Sometimes, you might want to sort your data depending on a specific property of your recording channels.

For example, when using multiple tetrodes, a good idea is to sort each tetrode separately. In this case, channels belonging to the same tetrode will be in the same 'group'. Alternatively, for long silicon probes, such as Neuropixels, you could sort different areas separately, for example hippocampus and thalamus.

All this can be done by sorting by 'property'. Properties can be loaded to the recording channels either manually (using the `set_channel_property` method), or by using a probe file. In this example we will create a 16 channel recording and split it in four channel groups (tetrodes).

Let's create a toy example with 16 channels (the `dumpable=True` dumps the extractors to a file, which is required for parallel sorting):

```
recording_tetrodes, sorting_true = se.example_datasets.toy_example(duration=10, num_
↪channels=16, dumpable=True)
```

Initially there is no group information ('location' is loaded automatically when creating toy data):

```
print(recording_tetrodes.get_shared_channel_property_names())
```

Out:

```
['gain', 'group', 'location', 'offset']
```

The file tetrode_16.prb contain the channel group description

```
channel_groups = {
    0: {
        'channels': [0,1,2,3],
    },
    1: {
        'channels': [4,5,6,7],
    },
    2: {
        'channels': [8,9,10,11],
    },
    3: {
        'channels': [12,13,14,15],
    }
}
```

We can load 'group' information using the '.prb' file:

```
recording_tetrodes = recording_tetrodes.load_probe_file('tetrode_16.prb')
print(recording_tetrodes.get_shared_channel_property_names())
```

Out:

```
['gain', 'group', 'location', 'offset']
```

We can now use the launcher to spike sort by the property 'group'. Internally, the recording is split into `SubRecordingExtractor` objects, one for each group. Each of them is spike sorted separately, yielding as many `SortingExtractor` objects as the number of groups. Finally, the sorting extractor objects are re-assembled into a single `MultiSortingExtractor`.

The different groups can also be sorted in parallel, and the output sorting extractor will have the same property used for sorting. Running in parallel (in separate threads) can speed up the computations.

Let's first run the four channel groups sequentially:

```
t_start = time.time()
sorting_tetrodes = ss.run_sorter('klusta', recording_tetrodes, output_folder='tmp_
↪tetrodes',
                                 grouping_property='group', parallel=False,␣
↪verbose=False)
print('Elapsed time: ', time.time() - t_start)
```

Out:

```
RUNNING SHELL SCRIPT: /home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/
↪checkouts/0.13.0/examples/modules/sorters/tmp_tetrodes/0/run_klusta.sh
/home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/checkouts/0.13.0/doc/
↪sources/spikesorters/spikesorters/basesorter.py:158: ResourceWarning: unclosed file
↪<_io.TextIOWrapper name=63 encoding='UTF-8'>
  self._run(recording, self.output_folders[i])
RUNNING SHELL SCRIPT: /home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/
↪checkouts/0.13.0/examples/modules/sorters/tmp_tetrodes/1/run_klusta.sh
RUNNING SHELL SCRIPT: /home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/
↪checkouts/0.13.0/examples/modules/sorters/tmp_tetrodes/2/run_klusta.sh
RUNNING SHELL SCRIPT: /home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/
↪checkouts/0.13.0/examples/modules/sorters/tmp_tetrodes/3/run_klusta.sh
Elapsed time:  7.512170314788818
```

then in parallel:

```
t_start = time.time()
sorting_tetrodes_p = ss.run_sorter('klusta', recording_tetrodes, output_folder='tmp_
↪tetrodes_par',
                                   grouping_property='group', parallel=True,␣
↪verbose=False)
print('Elapsed time parallel: ', time.time() - t_start)
```

Out:

```
Elapsed time parallel:  5.452665090560913
```

The units of the sorted output will have the same property used for spike sorting:

```
print(sorting_tetrodes.get_shared_unit_property_names())
```

Out:

```
['group', 'quality']
```

Note that channels can be split by any property. Let's for example assume that half of the tetrodes are in hippocampus CA1 region, and the other half is in CA3. first we have to load this property (this can be done also from the '.prb' file):

```
for ch in recording_tetrodes.get_channel_ids()[:int(recording_tetrodes.get_num_
↪channels() / 2)]:
    recording_tetrodes.set_channel_property(ch, property_name='region', value='CA1')

for ch in recording_tetrodes.get_channel_ids()[int(recording_tetrodes.get_num_
↪channels() / 2):]:
    recording_tetrodes.set_channel_property(ch, property_name='region', value='CA3')

for ch in recording_tetrodes.get_channel_ids():
    print(recording_tetrodes.get_channel_property(ch, property_name='region'))
```

Out:

```
CA1
CA1
CA1
CA1
CA1
CA1
CA1
CA1
CA3
CA3
CA3
CA3
CA3
CA3
CA3
CA3
```

Now let's spike sort by 'region' and check that the units of the sorted output have this property:

```
sorting_regions = ss.run_sorter('klusta', recording_tetrodes, output_folder='tmp_
↪regions',
                                grouping_property='region', parallel=True)

print(sorting_regions.get_shared_unit_property_names())
```

Out:

```
['group', 'quality', 'region']
```

Total running time of the script: (0 minutes 18.472 seconds)

6.3.4 Run spike sorting on concatenated recordings

In several experiments, several recordings are performed in sequence, for example a baseline/intervention. In these cases, since the underlying spiking activity can be assumed to be the same (or at least very similar), the recordings can be concatenated. This notebook shows how to concatenate the recordings before spike sorting and how to split the sorted output based on the concatenation.

```
import spikeinterface.extractors as se
import spikeinterface.sorters as ss
import time
```

When performing an experiment with multiple consecutive recordings, it can be a good idea to concatenate the single recordings, as this can improve the spike sorting performance and it doesn't require to track the neurons over the different recordings.

This can be done very easily in SpikeInterface using a combination of the `MultiRecordingTimeExtractor` and the `SubSortingExtractor` objects.

Let's create a toy example with 4 channels (the `dumpable=True` dumps the extractors to a file, which is required for parallel sorting):

```
recording_single, _ = se.example_datasets.toy_example(duration=10, num_channels=4,␣
↪dumpable=True)
```

Let's now assume that we have 4 recordings. In our case we will concatenate the `recording_single` 4 times. We first need to build a list of `RecordingExtractor` objects:

```
recordings_list = []
for i in range(4):
    recordings_list.append(recording_single)
```

We can now use the `recordings_list` to instantiate a `MultiRecordingTimeExtractor`, which concatenates the traces in time:

```
multirecording = se.MultiRecordingTimeExtractor(recordings=recordings_list)
```

Since the `MultiRecordingTimeExtractor` is a `RecordingExtractor`, we can run spike sorting "normally"

```
multisorting = ss.run_klusta(multirecording)
```

Out:

```
RUNNING SHELL SCRIPT: /home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/
↪checkouts/0.13.0/examples/modules/sorters/klusta_output/run_klusta.sh
/home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/checkouts/0.13.0/doc/
↪sources/spikesorters/spikesorters/basesorter.py:158: ResourceWarning: unclosed file
↪<_io.TextIOWrapper name=63 encoding='UTF-8'>
  self._run(recording, self.output_folders[i])
```

The returned `multisorting` object is a normal `SortingExtractor`, but we now that its spike trains are concatenated similarly to the recording concatenation. So we have to split them back. We can do that using the *epoch* information in the `MultiRecordingTimeExtractor`:

```
sortings = []

sortings = []
for epoch in multisorting.get_epoch_names():
    info = multisorting.get_epoch_info(epoch)
    sorting_single = se.SubSortingExtractor(multisorting, start_frame=info['start_
↪frame'], end_frame=info['end_frame'])
    sortings.append(sorting_single)
```

The `SortingExtractor` objects in the `sortings` list contain now split spike trains. The nice thing of this approach is that the unit_ids for the different epochs are the same unit!

Total running time of the script: (0 minutes 5.406 seconds)

6.4 Comparison tutorials

The `comparison` module imports the `spikecomparison` package. It allows to compare spike sorting output with and without ground-truth information.

6.4.1 Compare two sorters

This example show how to compare the result of two sorters.

Import

```
import numpy as np
import matplotlib.pyplot as plt

import spikeinterface.extractors as se
import spikeinterface.sorters as ss
import spikeinterface.comparison as sc
import spikeinterface.widgets as sw
```

First, let's create a toy example:

```
recording, sorting = se.example_datasets.toy_example(num_channels=4, duration=10,␣
↪seed=0)
```

Then run two spike sorters and compare their ouput.

```
sorting_KL = ss.run_klusta(recording)
sorting_MS4 = ss.run_mountainsort4(recording)
```

Out:

```
RUNNING SHELL SCRIPT: /home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/
↪checkouts/0.13.0/examples/modules/comparison/klusta_output/run_klusta.sh
/home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/checkouts/0.13.0/doc/
↪sources/spikesorters/spikesorters/basesorter.py:158: ResourceWarning: unclosed file
↪<_io.TextIOWrapper name=63 encoding='UTF-8'>
  self._run(recording, self.output_folders[i])
Warning! The recording is already filtered, but Mountainsort4 filter is enabled. You␣
↪can disable filters by setting 'filter' parameter to False
```

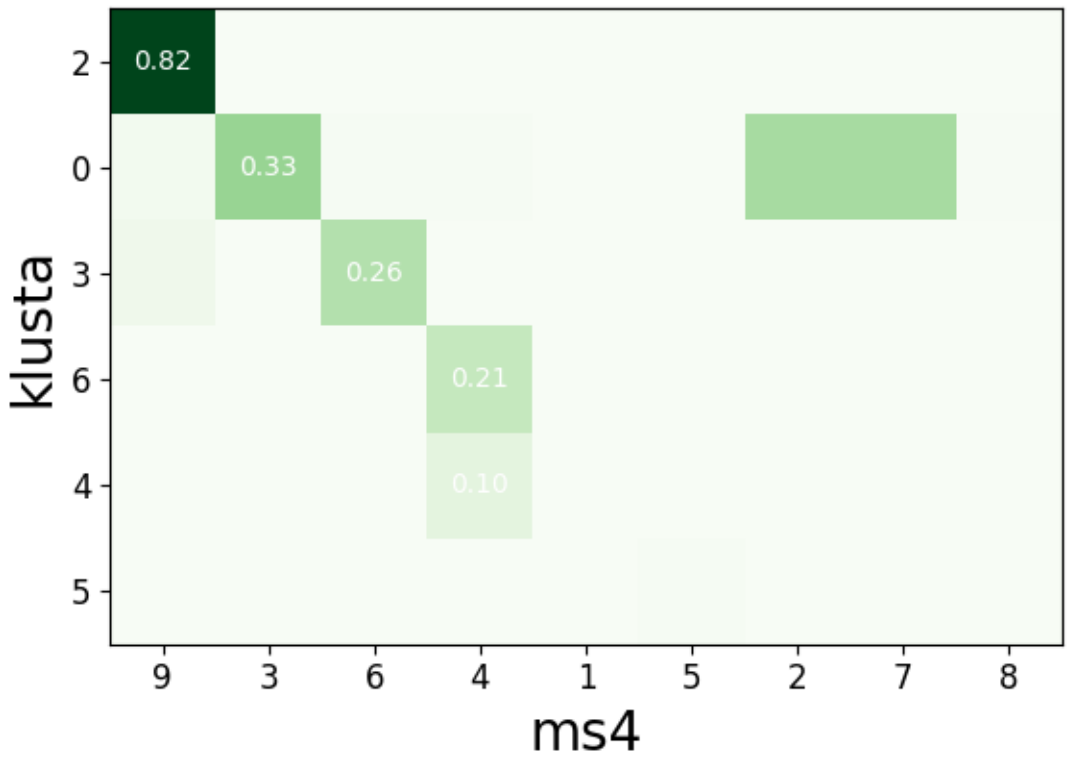
The `compare_two_sorters` function allows us to compare the spike sorting output. It returns a `SortingComparison` object, with methods to inspect the comparison output easily. The comparison matches the units by comparing the agreement between unit spike trains.

Let's see how to inspect and access this matching.

```
cmp_KL_MS4 = sc.compare_two_sorters(sorting1=sorting_KL, sorting2=sorting_MS4,
                                                    sorting1_name='klusta', sorting2_name=
↪'ms4')
```

We can check the agreement matrix to inspect the matching.

```
sw.plot_agreement_matrix(cmp_KL_MS4)
```



Out:

```
<spikewidgets.widgets.agreementmatrixwidget.agreementmatrixwidget.
↪AgreementMatrixWidget object at 0x7f3dfc30a160>
```

Some useful internal dataframes help to check the match and count like **match_event_count** or **agreement_scores**

```
print(cmp_KL_MS4.match_event_count)
print(cmp_KL_MS4.agreement_scores)
```

Out:

```
   1   2   3   4  5   6   7  8   9
0  0  22  25   1  0   1  22  2   3
2  0   0   0   0  0   0   0  0  27
3  0   0   0   0  0  11   0  0   2
4  0   0   0   8  0   0   0  0   0
5  0   0   0   0  1   0   0  0   0
6  0   0   0  16  0   0   0  0   0
     1         2         3         4  ...         6         7         8         9
0  0.0  0.289474  0.328947  0.006579  ...  0.008621  0.289474  0.005865  0.028302
2  0.0  0.000000  0.000000  0.000000  ...  0.000000  0.000000  0.000000  0.818182
3  0.0  0.000000  0.000000  0.000000  ...  0.255814  0.000000  0.000000  0.045455
4  0.0  0.000000  0.000000  0.103896  ...  0.000000  0.000000  0.000000  0.000000
5  0.0  0.000000  0.000000  0.000000  ...  0.000000  0.000000  0.000000  0.000000
6  0.0  0.000000  0.000000  0.207792  ...  0.000000  0.000000  0.000000  0.000000
```

(continues on next page)

(continued from previous page)

```
[6 rows x 9 columns]
```

In order to check which units were matched, the `get_mapped_sorting` methods can be used. If units are not matched they are listed as -1.

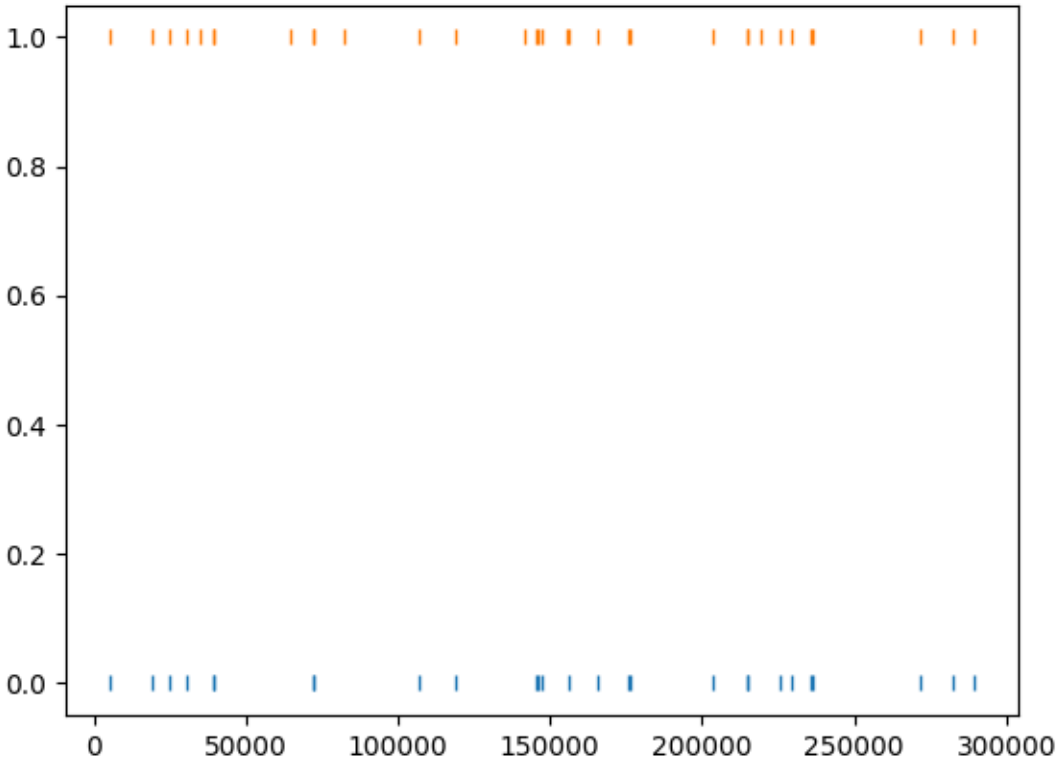
```
# units matched to klusta units
mapped_sorting_klusta = cmp_KL_MS4.get_mapped_sorting1()
print('Klusta units:', sorting_KL.get_unit_ids())
print('Klusta mapped units:', mapped_sorting_klusta.get_mapped_unit_ids())

# units matched to ms4 units
mapped_sorting_ms4 = cmp_KL_MS4.get_mapped_sorting2()
print('Mountainsort units:',sorting_MS4.get_unit_ids())
print('Mountainsort mapped units:',mapped_sorting_ms4.get_mapped_unit_ids())
```

Out:

```
Klusta units: [0, 2, 3, 4, 5, 6]
Klusta mapped units: [-1, 9, -1, -1, -1, -1]
Mountainsort units: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Mountainsort mapped units: [-1, -1, -1, -1, -1, -1, -1, -1, 2]
```

The :code:get_unit_spike_train‘ returns the mapped spike train. We can use it to check the spike times.

```
# find a unit from KL that have a match
ind  = np.where(np.array(mapped_sorting_klusta.get_mapped_unit_ids())!=-1)[0][0]
u1 = sorting_KL.get_unit_ids()[ind]
print(ind, u1)

# check that matched spike trains correspond
st1 = sorting_KL.get_unit_spike_train(u1)
st2 = mapped_sorting_klusta.get_unit_spike_train(u1)
fig, ax = plt.subplots()
ax.plot(st1, np.zeros(st1.size), '|')
ax.plot(st2, np.ones(st2.size), '|')

plt.show()
```



Out:

```
1 2
```

Total running time of the script: (0 minutes 3.869 seconds)

6.4.2 Compare multiple sorters and consensus based method

With 3 or more spike sorters, the comparison is implemented with a graph-based method. The multiple sorter comparison also allows to clean the output by applying a consensus-based method which only selects spike trains and spikes in agreement with multiple sorters.

Import

```
import numpy as np
import matplotlib.pyplot as plt

import spikeinterface.extractors as se
import spikeinterface.sorters as ss
import spikeinterface.comparison as sc
import spikeinterface.widgets as sw
```

First, let's create a toy example:

```
recording, sorting = se.example_datasets.toy_example(num_channels=4, duration=20,␣
↪seed=0)
```

Then run 3 spike sorters and compare their output.

```
sorting_KL = ss.run_klusta(recording)
sorting_MS4 = ss.run_mountainsort4(recording)
sorting_TDC = ss.run_tridesclous(recording)
```

Out:

```
RUNNING SHELL SCRIPT: /home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/
↪checkouts/0.13.0/examples/modules/comparison/klusta_output/run_klusta.sh
/home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/checkouts/0.13.0/doc/
↪sources/spikesorters/spikesorters/basesorter.py:158: ResourceWarning: unclosed file
↪<_io.TextIOWrapper name=63 encoding='UTF-8'>
  self._run(recording, self.output_folders[i])
Warning! The recording is already filtered, but Mountainsort4 filter is enabled. You␣
↪can disable filters by setting 'filter' parameter to False
{'chunksize': 3000,
 'clean_cluster': {'apply_auto_merge_cluster': True,
                   'apply_auto_split': True,
                   'apply_trash_low_extremum': True,
                   'apply_trash_not_aligned': True,
                   'apply_trash_small_cluster': True},
 'clean_peaks': {'alien_value_threshold': None, 'mode': 'extremum_amplitude'},
 'cluster_kargs': {'adjacency_radius_um': 0.0,
                   'high_adjacency_radius_um': 0.0,
                   'max_loop': 1000,
                   'min_cluster_size': 20},
 'cluster_method': 'pruningshears',
 'duration': 20.0,
 'extract_waveforms': {'wf_left_ms': -2.0, 'wf_right_ms': 3.0},
 'feature_kargs': {'n_components': 8},
 'feature_method': 'global_pca',
 'make_catalogue': {'inter_sample_oversampling': False,
                    'sparse_thresh_level2': 3,
                    'subsample_ratio': 'auto'},
 'memory_mode': 'memmap',
 'mode': 'dense',
 'n_jobs': -1,
 'n_spike_for_centroid': 350,
 'noise_snippet': {'nb_snippet': 300},
 'peak_detector': {'adjacency_radius_um': 200.0,
                   'engine': 'numpy',
                   'method': 'global',
                   'peak_sign': '-',
                   'peak_span_ms': 0.7,
                   'relative_threshold': 5,
                   'smooth_radius_um': None},
 'peak_sampler': {'mode': 'rand', 'nb_max': 20000, 'nb_max_by_channel': None},
 'preprocessor': {'common_ref_removal': False,
                  'engine': 'numpy',
                  'highpass_freq': 400.0,
                  'lostfront_chunksize': -1,
                  'lowpass_freq': 5000.0,
                  'smooth_size': 0},
 'sparse_threshold': 1.5}
```

Compare multiple spike sorter outputs

```
mcmp = sc.compare_multiple_sorters(sorting_list=[sorting_KL, sorting_MS4, sorting_
↪TDC],
                                   name_list=['KL', 'MS4', 'TDC'], verbose=True)
```

Out:

```
Multicomaprison step 1: pairwise comparison
  Comparing:  KL  and  MS4
  Comparing:  KL  and  TDC
  Comparing:  MS4  and  TDC
Multicomaprison step 2: make graph
Multicomaprison step 3: clean graph
Removed 0 duplicate nodes
Multicomaprison step 4: extract agreement from graph
```

The multiple sorters comparison internally computes pairwise comparison, that can be accessed as follows:

```
print(mcmp.comparisons[0].sorting1, mcmp.comparisons[0].sorting2)
mcmp.comparisons[0].get_mapped_sorting1().get_mapped_unit_ids()
```

Out:

```
<spikeextractors.extractors.klustaextractors.klustaextractors.KlustaSortingExtractor␣
↪object at 0x7f3dfc303a20> <spikeextractors.extractors.mdaextractors.mdaextractors.
↪MdaSortingExtractor object at 0x7f3e05685fd0>

[6, 10, 3, 9, -1, -1, -1]
```

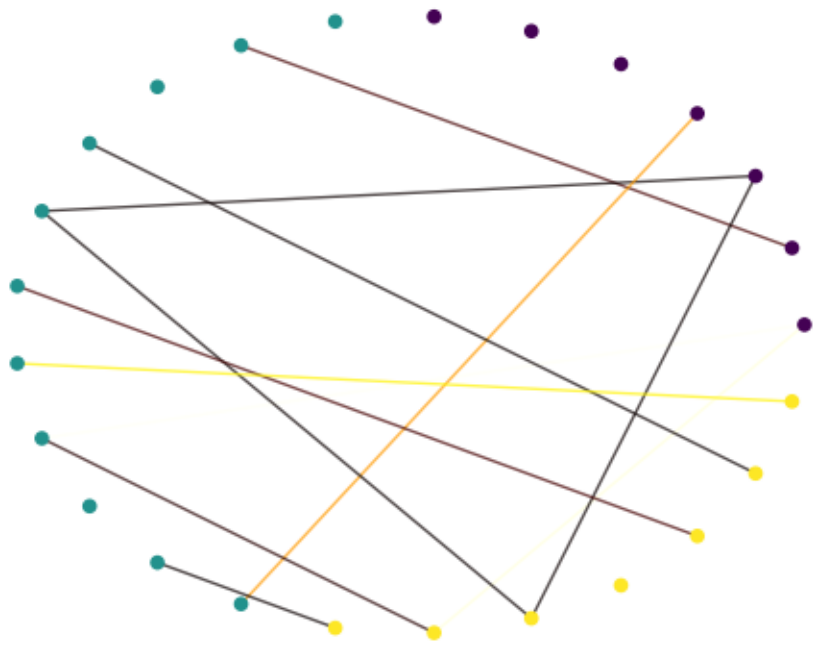
```
print(mcmp.comparisons[1].sorting1, mcmp.comparisons[1].sorting2)
mcmp.comparisons[0].get_mapped_sorting1().get_mapped_unit_ids()
```

Out:

```
<spikeextractors.extractors.klustaextractors.klustaextractors.KlustaSortingExtractor␣
↪object at 0x7f3dfc303a20> <spikeextractors.extractors.tridescloussortingextractor.
↪tridescloussortingextractor.TridesclousSortingExtractor object at 0x7f3dfc25f7f0>

[6, 10, 3, 9, -1, -1, -1]
```

The global multi comparison can be visualized with this graph

```
sw.plot_multicomp_graph(mcmp)
```



Out:

```
<spikewidgets.widgets.multicompgraphwidget.multicompgraphwidget.MultiCompGraphWidget␣
↪object at 0x7f3dfc25f7b8>
```

We can see that there is a better agreement between tridesclous and mountainsort (5 units matched), while klusta only has two matched units with tridesclous, and three with mountainsort.

Consensus-based method

We can pull the units in agreement with different sorters using the `get_agreement_sorting` method. This allows to make spike sorting more robust by integrating the output of several algorithms. On the other hand, it might suffer from weak performance of single algorithms.

When extracting the units in agreement, the spike trains are modified so that only the true positive spikes between the comparison with the best match are used.

```
agr_3 = mcmp.get_agreement_sorting(minimum_agreement_count=3)
print('Units in agreement for all three sorters: ', agr_3.get_unit_ids())
```

Out:

```
Units in agreement for all three sorters:  [0, 2]
```

```
agr_2 = mcmp.get_agreement_sorting(minimum_agreement_count=2)
print('Units in agreement for at least two sorters: ', agr_2.get_unit_ids())
```

Out:

```
Units in agreement for at least two sorters:  [0, 1, 2, 3, 8, 9, 10, 12]
```

```
agr_all = mcmp.get_agreement_sorting()
```

The unit index of the different sorters can also be retrieved from the agreement sorting object (`agr_3`) property `sorter_unit_ids`.

```
print(agr_3.get_shared_unit_property_names())
```

Out:

```
['agreement_number', 'avg_agreement', 'sorter_unit_ids']
```

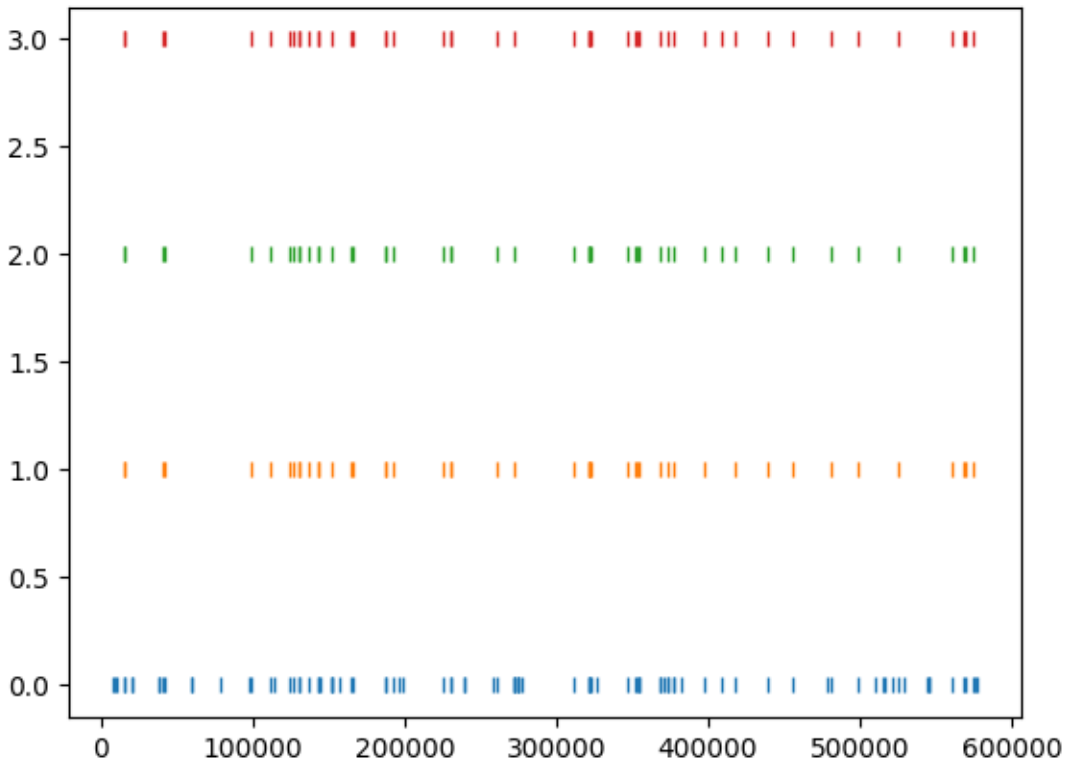
```
print(agr_3.get_unit_ids())
# take one unit in agreement
u = agr_3.get_unit_ids()[0]
print(agr_3.get_unit_property(u, 'sorter_unit_ids'))
```

Out:

```
[0, 2]
{'MS4': 6, 'KL': 0, 'TDC': 1}
```

Now that we found our unit, we can plot a rasters with the spike trains of the single sorters and the one from the consensus based method. When extracting the agreement sorting, spike trains are cleaned so that only true positives remain from the comparison with the largest agreement are kept. Let's take a look at the raster plots for the different sorters and the agreement sorter:

```
d = agr_3.get_unit_property(u, 'sorter_unit_ids')
st0 = sorting_KL.get_unit_spike_train(d['KL'])
st1 = sorting_MS4.get_unit_spike_train(d['MS4'])
st2 = sorting_TDC.get_unit_spike_train(d['TDC'])
st3 = agr_3.get_unit_spike_train(u)

fig, ax = plt.subplots()
ax.plot(st0, 0 * np.ones(st0.size), '|')
ax.plot(st1, 1 * np.ones(st1.size), '|')
ax.plot(st2, 2 * np.ones(st2.size), '|')
ax.plot(st3, 3 * np.ones(st3.size), '|')

print('Klusta spike train length', st0.size)
print('Mountainsort spike train length', st1.size)
print('Tridesclous spike train length', st2.size)
print('Agreement spike train length', st3.size)
```



Out:

```
Klusta spike train length 93
Mountainsort spike train length 47
Tridesclous spike train length 48
Agreement spike train length 48
```

As we can see, the best match is between Mountainsort and Tridesclous, but only the true positive spikes make up the agreement spike train.

```
plt.show()
```

Total running time of the script: (0 minutes 7.536 seconds)

6.4.3 Compare spike sorting output with ground-truth recordings

Simulated recordings or paired pipette and extracellular recordings can be used to validate spike sorting algorithms.

For comparing to ground-truth data, the `compare_sorter_to_ground_truth(gt_sorting, tested_sorting)` function can be used. In this recording, we have ground-truth information for all units, so we can set `exhaustive_gt` to `True`.

Import

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

import spikeinterface.extractors as se
import spikeinterface.sorters as ss
import spikeinterface.comparison as sc
import spikeinterface.widgets as sw
```

```
recording, sorting_true = se.example_datasets.toy_example(num_channels=4, duration=10,
↪ seed=0)

sorting_MS4 = ss.run_mountainsort4(recording)
```

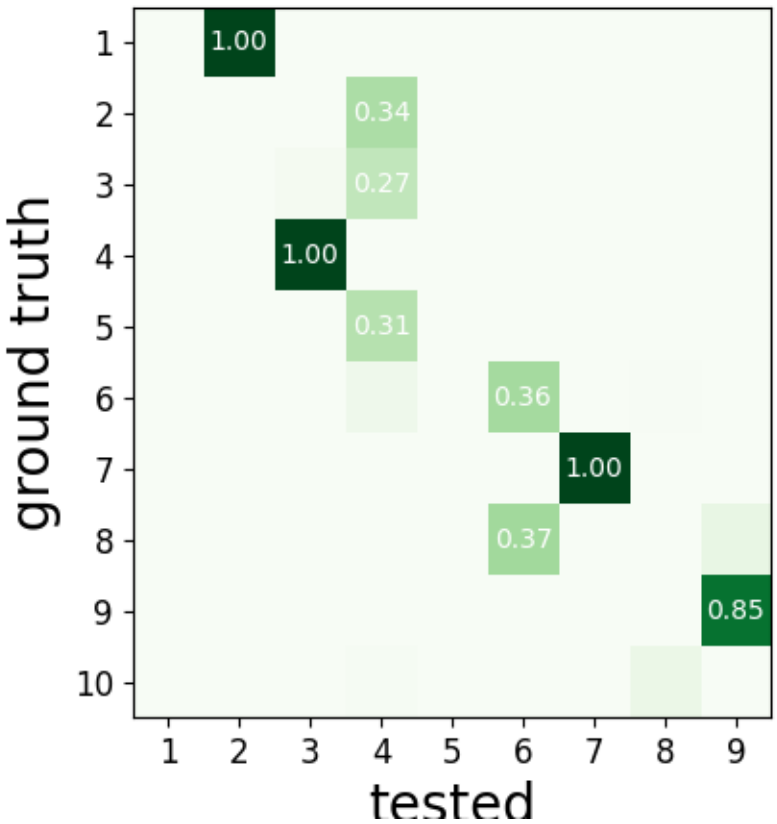
Out:

```
Warning! The recording is already filtered, but Mountainsort4 filter is enabled. You␣
↪can disable filters by setting 'filter' parameter to False
```

```
cmp_gt_MS4 = sc.compare_sorter_to_ground_truth(sorting_true, sorting_MS4, exhaustive_
↪gt=True)
```

To have an overview of the match we can use the unordered agreement matrix

```
sw.plot_agreement_matrix(cmp_gt_MS4, ordered=False)
```

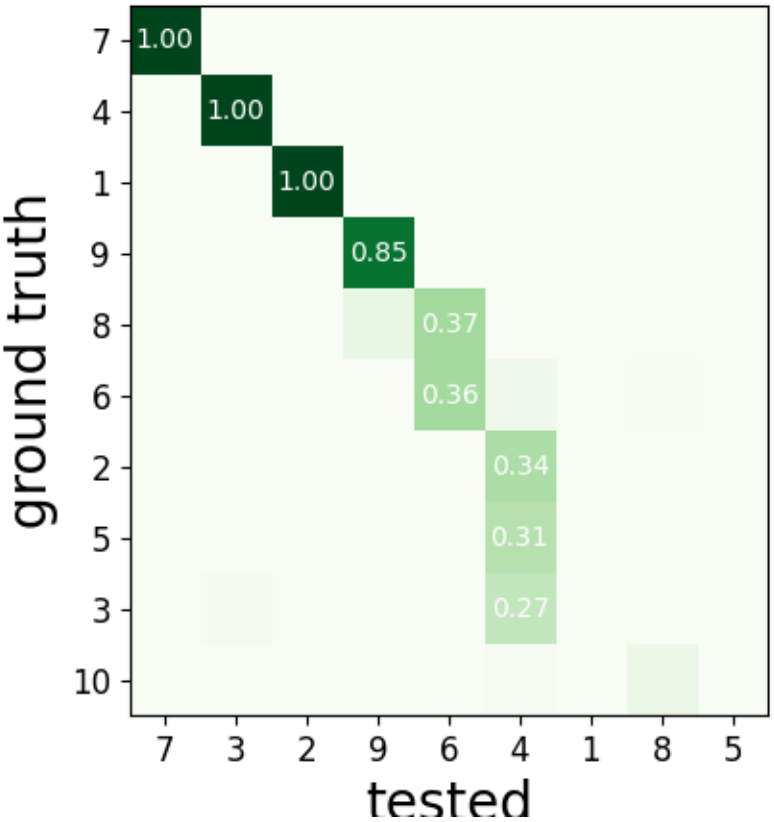


Out:

```
<spikewidgets.widgets.agreementmatrixwidget.agreementmatrixwidget.
↪AgreementMatrixWidget object at 0x7f3dfc2ef978>
```

or ordered

```
sw.plot_agreement_matrix(cmp_gt_MS4, ordered=True)
```



Out:

```
<spikewidgets.widgets.agreementmatrixwidget.agreementmatrixwidget.
↪AgreementMatrixWidget object at 0x7f3dfc73f0b8>
```

This function first matches the ground-truth and spike sorted units, and then it computes several performance metrics.

Once the spike trains are matched, each spike is labelled as:

- true positive (tp): spike found both in `gt_sorting` and `tested_sorting`
- false negative (fn): spike found in `gt_sorting`, but not in `tested_sorting`
- false positive (fp): spike found in `tested_sorting`, but not in `gt_sorting`

From the counts of these labels the following performance measures are computed:

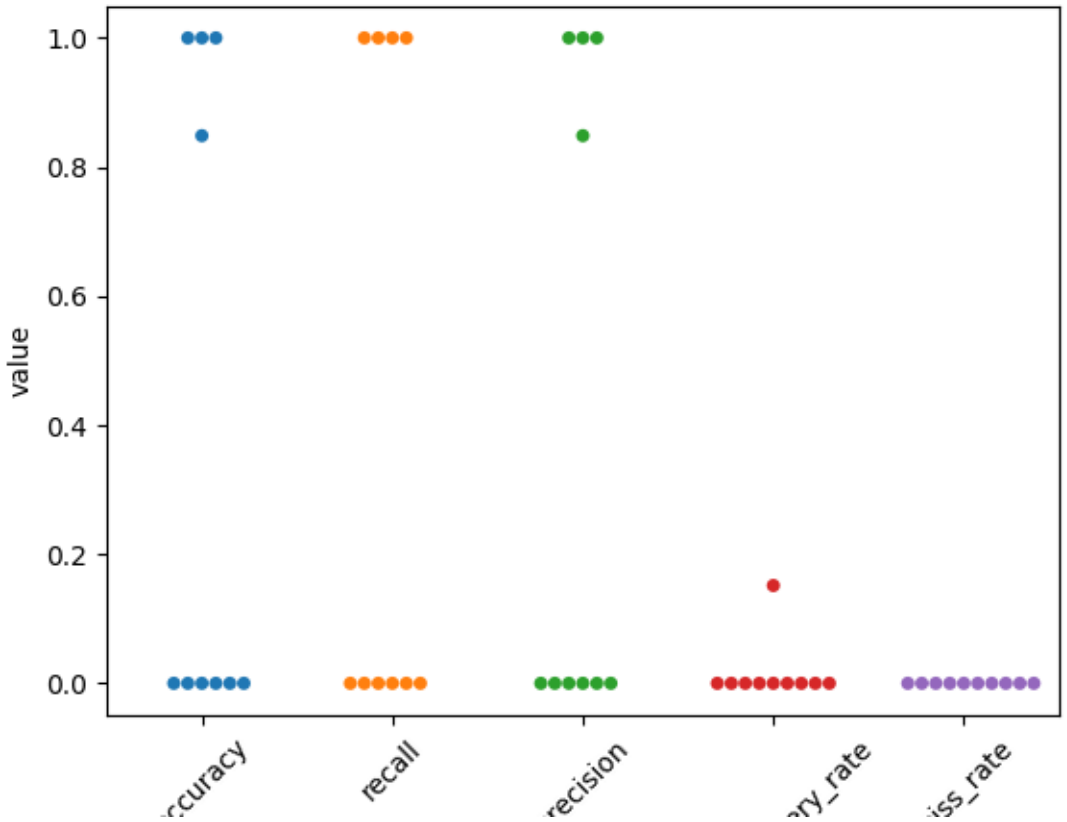
- accuracy: #tp / (#tp+ #fn + #fp)
- recall: #tp / (#tp + #fn)
- precision: #tp / (#tp + #fn)
- miss rate: #fn / (#tp + #fn1)
- false discovery rate: #fp / (#tp + #fp)

The `get_performance` method a pandas dataframe (or a dictionary if `output='dict'`) with the comparison metrics. By default, these are calculated for each spike train of `sorting1:code:`, the results can be pooles by average (average of the metrics) and by sum (all counts are summed and the metrics are computed then).

```
perf = cmp_gt_MS4.get_performance()
```

Lets use seaborn swarm plot

```
fig1, ax1 = plt.subplots()
perf2 = pd.melt(perf, var_name='measurement')
ax1 = sns.swarmplot(data=perf2, x='measurement', y='value', ax=ax1)
ax1.set_xticklabels(labels=ax1.get_xticklabels(), rotation=45)
```



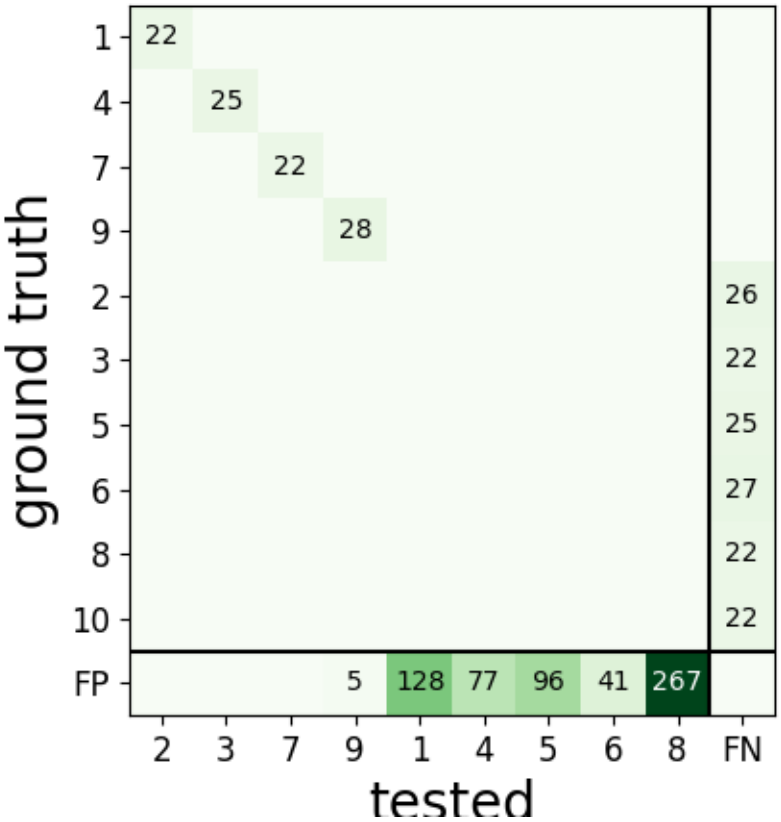
Out:

```
[Text(0, 0, 'accuracy'), Text(1, 0, 'recall'), Text(2, 0, 'precision'), Text(3, 0,
↪'false_discovery_rate'), Text(4, 0, 'miss_rate')]
```

The confusion matrix is also a good summary of the score as it has the same shape as agreement matrix, but it contains an extra column for FN

and an extra row for FP

```
sw.plot_confusion_matrix(cmp_gt_MS4)
```



Out:

```
<spikewidgets.widgets.confusionmatrixwidget.confusionmatrixwidget.
↪ConfusionMatrixWidget object at 0x7f3e056777b8>
```

We can query the well and bad detected units. By default, the threshold on accuracy is 0.95.

```
cmp_gt_MS4.get_well_detected_units()
```

Out:

```
[2, 3, 7, 9]
```

```
cmp_gt_MS4.get_false_positive_units()
```

Out:

```
[1, 5, 8]
```

```
cmp_gt_MS4.get_redundant_units()
```

Out:

```
[]
```

Lets do the same for klusta

```
sorting_KL = ss.run_klusta(recording)
cmp_gt_KL = sc.compare_sorter_to_ground_truth(sorting_true, sorting_KL, exhaustive_
↪gt=True)
```

Out:

```
RUNNING SHELL SCRIPT: /home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/
↪checkouts/0.13.0/examples/modules/comparison/klusta_output/run_klusta.sh
/home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/checkouts/0.13.0/doc/
↪sources/spikesorters/spikesorters/basesorter.py:158: ResourceWarning: unclosed file
↪<_io.TextIOWrapper name=63 encoding='UTF-8'>
  self._run(recording, self.output_folders[i])
```

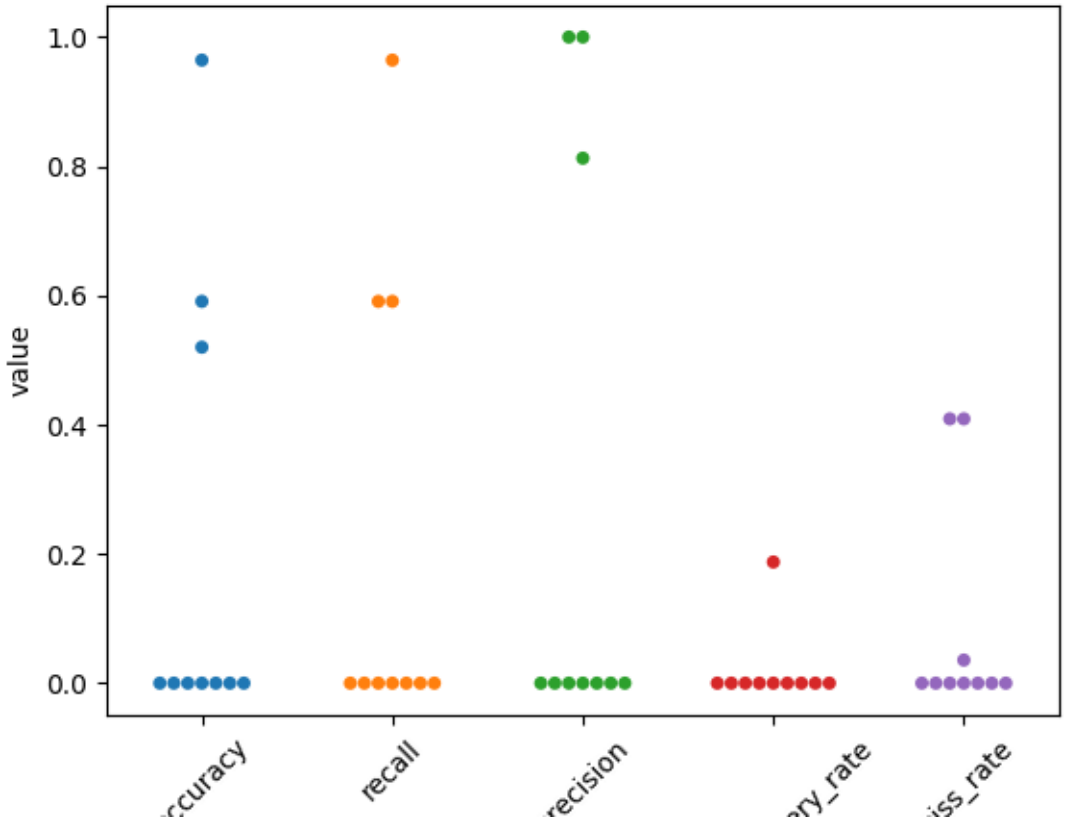
```
perf = cmp_gt_KL.get_performance()

print(perf)
```

Out:

gt_unit_id	accuracy	recall	precision	false_discovery_rate	miss_rate
1	0	0	0	0	0
2	0	0	0	0	0
3	0.52	0.590909	0.8125	0.1875	0.409091
4	0	0	0	0	0
5	0	0	0	0	0
6	0	0	0	0	0
7	0	0	0	0	0
8	0.590909	0.590909	1	0	0.409091
9	0.964286	0.964286	1	0	0.0357143
10	0	0	0	0	0

Lets use seaborn swarm plot

```
fig2, ax2 = plt.subplots()
perf2 = pd.melt(perf, var_name='measurement')
ax2 = sns.swarmplot(data=perf2, x='measurement', y='value', ax=ax2)
ax2.set_xticklabels(labels=ax2.get_xticklabels(), rotation=45)
```



Out:

```
[Text(0, 0, 'accuracy'), Text(1, 0, 'recall'), Text(2, 0, 'precision'), Text(3, 0,
↪'false_discovery_rate'), Text(4, 0, 'miss_rate')]
```

```
print(cmp_gt_KL.get_well_detected_units)
```

Out:

```
<bound method GroundTruthComparison.get_well_detected_units of <spikecomparison.
↪groundtruthcomparison.GroundTruthComparison object at 0x7f3dfc861cc0>>
```

```
print(cmp_gt_KL.get_false_positive_units())
```

Out:

```
[6]
```

```
print(cmp_gt_KL.get_redundant_units())
```

Out:

```
[]
```

Total running time of the script: (0 minutes 4.533 seconds)

6.4.4 Ground truth study tutorial

This tutorial illustrates how to run a "study". A study is a systematic performance comparisons several ground truth recordings with several sorters.

The submodule study and the class propose high level tools functions to run many groundtruth comparison with many sorter on many recordings and then collect and aggregate results in an easy way.

The all mechanism is based on an intrinsinct organisation into a "study_folder" with several subfolder:

- raw_files : contain a copy in binary format of recordings
- sorter_folders : contains output of sorters
- ground_truth : contains a copy of sorting ground in npz format
- sortings: contains light copy of all sorting in npz format
- tables: some table in cvs format

In order to run and re run the computation all gt_sorting anf recordings are copied to a fast and universal format : binary (for recordings) and npz (for sortings).

Imports

```
import matplotlib.pyplot as plt
import seaborn as sns

import spikeinterface.extractors as se
import spikeinterface.widgets as sw
from spikeinterface.comparison import GroundTruthStudy
```

Setup study folder and run all sorters

We first generate the folder. this can take some time because recordings are copied inside the folder.

```
rec0, gt_sorting0 = se.example_datasets.toy_example(num_channels=4, duration=10,␣
↪seed=10)
rec1, gt_sorting1 = se.example_datasets.toy_example(num_channels=4, duration=10,␣
↪seed=0)
gt_dict = {
    'rec0': (rec0, gt_sorting0),
    'rec1': (rec1, gt_sorting1),
}
study_folder = 'a_study_folder'
study = GroundTruthStudy.create(study_folder, gt_dict)
```

Then just run all sorters on all recordings in one functions.

```
# sorter_list = st.sorters.available_sorters() # this get all sorters.
sorter_list = ['klusta', 'tridesclous', 'mountainsort4']
study.run_sorters(sorter_list, mode="keep")
```

Out:

```
RUNNING SHELL SCRIPT: /home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/
↪checkouts/0.13.0/examples/modules/comparison/a_study_folder/sorter_folders/rec0/
↪klusta/run_klusta.sh
/home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/checkouts/0.13.0/doc/
↪sources/spikesorters/spikesorters/basesorter.py:158: ResourceWarning: unclosed file
↪<_io.TextIOWrapper name=63 encoding='UTF-8'>
```

(continues on next page)

(continued from previous page)

```
  self._run(recording, self.output_folders[i])
{'chunksize': 3000,
 'clean_cluster': {'apply_auto_merge_cluster': True,
                   'apply_auto_split': True,
                   'apply_trash_low_extremum': True,
                   'apply_trash_not_aligned': True,
                   'apply_trash_small_cluster': True},
 'clean_peaks': {'alien_value_threshold': None, 'mode': 'extremum_amplitude'},
 'cluster_kargs': {'adjacency_radius_um': 0.0,
                   'high_adjacency_radius_um': 0.0,
                   'max_loop': 1000,
                   'min_cluster_size': 20},
 'cluster_method': 'pruningshears',
 'duration': 10.0,
 'extract_waveforms': {'wf_left_ms': -2.0, 'wf_right_ms': 3.0},
 'feature_kargs': {'n_components': 8},
 'feature_method': 'global_pca',
 'make_catalogue': {'inter_sample_oversampling': False,
                    'sparse_thresh_level2': 3,
                    'subsample_ratio': 'auto'},
 'memory_mode': 'memmap',
 'mode': 'dense',
 'n_jobs': -1,
 'n_spike_for_centroid': 350,
 'noise_snippet': {'nb_snippet': 300},
 'peak_detector': {'adjacency_radius_um': 200.0,
                   'engine': 'numpy',
                   'method': 'global',
                   'peak_sign': '-',
                   'peak_span_ms': 0.7,
                   'relative_threshold': 5,
                   'smooth_radius_um': None},
 'peak_sampler': {'mode': 'rand', 'nb_max': 20000, 'nb_max_by_channel': None},
 'preprocessor': {'common_ref_removal': False,
                  'engine': 'numpy',
                  'highpass_freq': 400.0,
                  'lostfront_chunksize': -1,
                  'lowpass_freq': 5000.0,
                  'smooth_size': 0},
 'sparse_threshold': 1.5}
RUNNING SHELL SCRIPT: /home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/
↪checkouts/0.13.0/examples/modules/comparison/a_study_folder/sorter_folders/rec1/
↪klusta/run_klusta.sh
/home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/checkouts/0.13.0/doc/
↪sources/spikesorters/spikesorters/basesorter.py:158: ResourceWarning: unclosed file
↪<_io.TextIOWrapper name=63 encoding='UTF-8'>
  self._run(recording, self.output_folders[i])
{'chunksize': 3000,
 'clean_cluster': {'apply_auto_merge_cluster': True,
                   'apply_auto_split': True,
                   'apply_trash_low_extremum': True,
                   'apply_trash_not_aligned': True,
                   'apply_trash_small_cluster': True},
 'clean_peaks': {'alien_value_threshold': None, 'mode': 'extremum_amplitude'},
 'cluster_kargs': {'adjacency_radius_um': 0.0,
                   'high_adjacency_radius_um': 0.0,
                   'max_loop': 1000,
```

(continues on next page)

(continued from previous page)

```
                   'min_cluster_size': 20},
 'cluster_method': 'pruningshears',
 'duration': 10.0,
 'extract_waveforms': {'wf_left_ms': -2.0, 'wf_right_ms': 3.0},
 'feature_kargs': {'n_components': 8},
 'feature_method': 'global_pca',
 'make_catalogue': {'inter_sample_oversampling': False,
                    'sparse_thresh_level2': 3,
                    'subsample_ratio': 'auto'},
 'memory_mode': 'memmap',
 'mode': 'dense',
 'n_jobs': -1,
 'n_spike_for_centroid': 350,
 'noise_snippet': {'nb_snippet': 300},
 'peak_detector': {'adjacency_radius_um': 200.0,
                   'engine': 'numpy',
                   'method': 'global',
                   'peak_sign': '-',
                   'peak_span_ms': 0.7,
                   'relative_threshold': 5,
                   'smooth_radius_um': None},
 'peak_sampler': {'mode': 'rand', 'nb_max': 20000, 'nb_max_by_channel': None},
 'preprocessor': {'common_ref_removal': False,
                  'engine': 'numpy',
                  'highpass_freq': 400.0,
                  'lostfront_chunksize': -1,
                  'lowpass_freq': 5000.0,
                  'smooth_size': 0},
 'sparse_threshold': 1.5}
```

You can re run **run_study_sorters** as many time as you want. By default **mode='keep'** so only uncomputed sorter are rerun. For instance, so just remove the "sorter_folders/rec1/herdingspikes" to re-run only one sorter on one recording.

Then we copy the spike sorting outputs into a separate subfolder. This allow to remove the "large" sorter_folders.

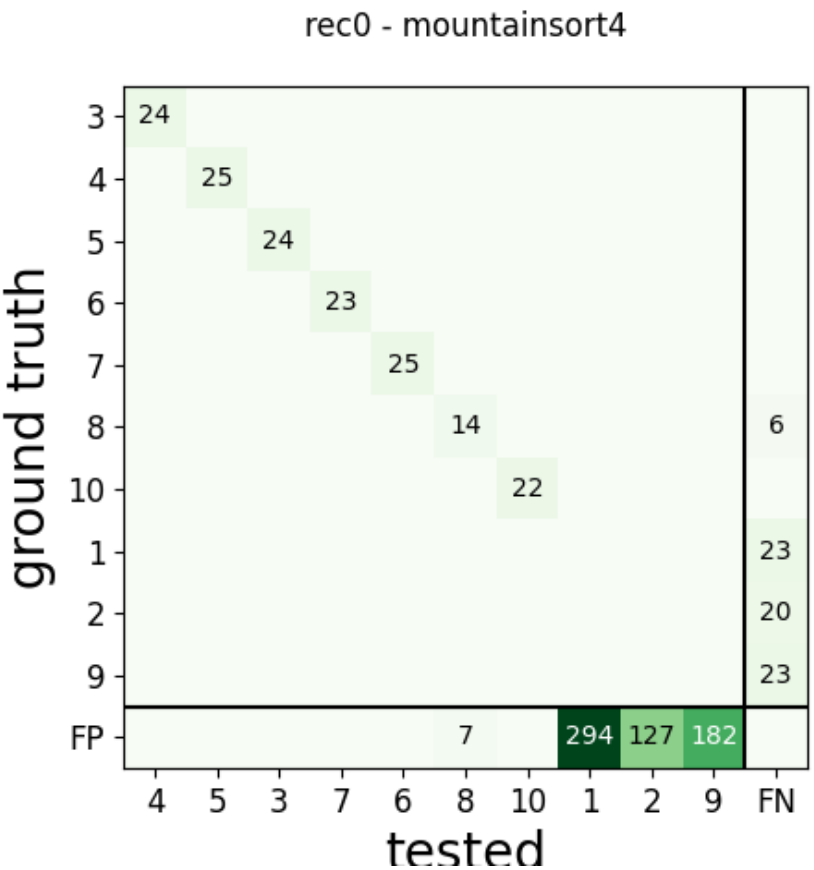
```
study.copy_sortings()
```

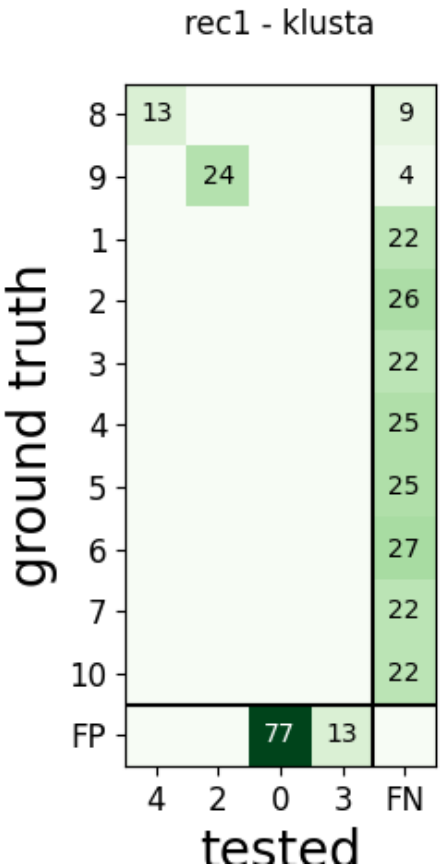
Collect comparisons

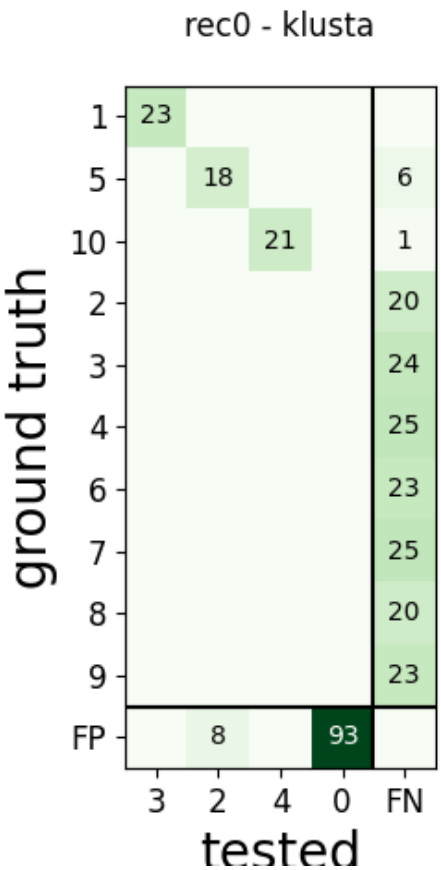
You can collect in one shot all results and run the GroundTruthComparison on it. So you can acces finely to all individual results.

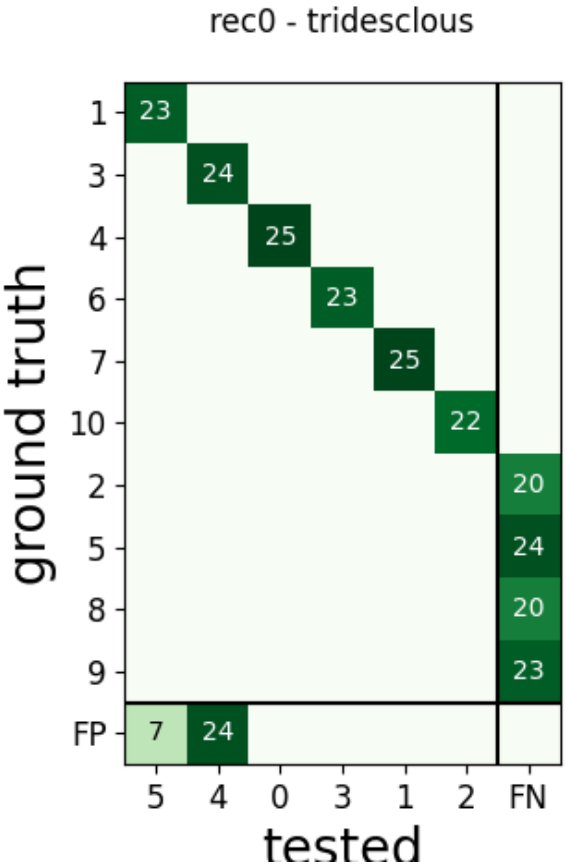
Note that exhaustive_gt=True when you excatly how many units in ground truth (for synthetic datasets)

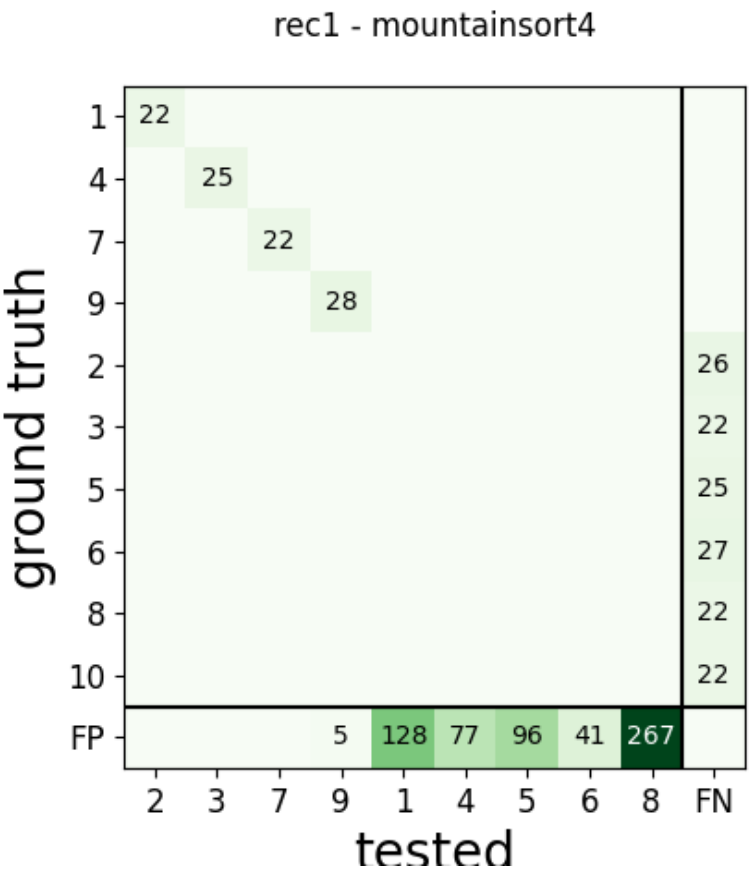
```
study.run_comparisons(exhaustive_gt=True)

for (rec_name, sorter_name), comp in study.comparisons.items():
    print('*' * 10)
    print(rec_name, sorter_name)
    print(comp.count_score)  # raw counting of tp/fp/...
    comp.print_summary()
    perf_unit = comp.get_performance(method='by_unit')
    perf_avg = comp.get_performance(method='pooled_with_average')
    m = comp.get_confusion_matrix()
    w_comp = sw.plot_confusion_matrix(comp)
    w_comp.ax.set_title(rec_name  + ' - ' + sorter_name)
```

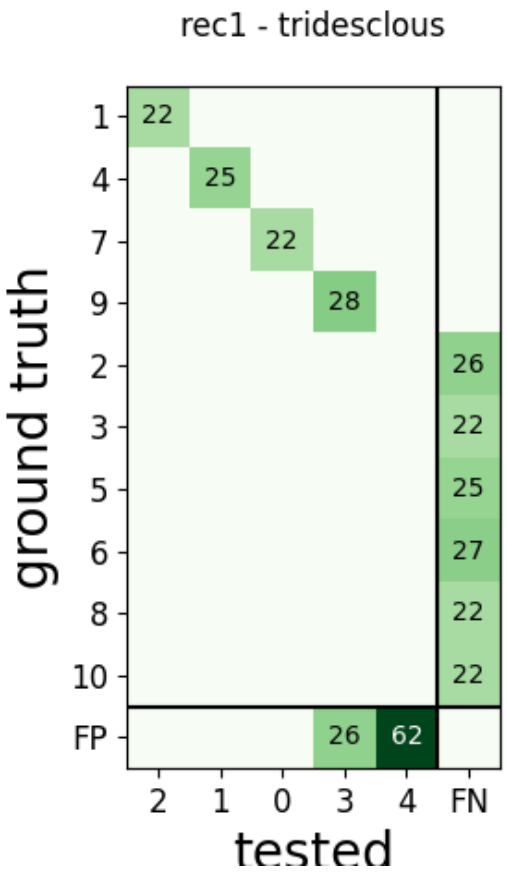

rec0 - mountainsort4
ground truth
tested
3
4
5
6
7
8
10
1
2
9
FP
4
5
3
7
6
8
10
1
2
9
FN
24
25
24
23
25
14
22
7
294
127
182
6
23
20
23


rec1 - klusta
ground truth
tested
8
9
1
2
3
4
5
6
7
10
FP
4
2
0
3
FN
13
24
9
4
22
26
22
25
25
27
22
22
77
13


rec0 - klusta
ground truth
tested
1
5
10
2
3
4
6
7
8
9
FP
3
2
4
0
FN
23
18
21
8
93
6
1
20
24
25
23
25
20
23


rec0 - tridesclous
ground truth
tested
1
3
4
6
7
10
2
5
8
9
FP
5
4
0
3
1
2
FN
23
24
25
23
25
22
20
24
20
23
7
24


rec1 - mountainsort4
ground truth
1
4
7
9
2
3
5
6
8
10
FP
22
25
22
28
26
22
25
27
22
22
5
128
77
96
41
267
2
3
7
9
1
4
5
6
8
FN
tested



Out:

```
**********
rec0 mountainsort4
             tp  fn fp num_gt num_tested tested_id
gt_unit_id
1             0  23  0     23          0        -1
2             0  20  0     20          0        -1
3            24   0  0     24         24         4
4            25   0  0     25         25         5
5            24   0  0     24         24         3
6            23   0  0     23         23         7
7            25   0  0     25         25         6
8            14   6  7     20         21         8
9             0  23  0     23          0        -1
10           22   0  0     22         22        10
SUMMARY
-------
GT num_units: 10
TESTED num_units: 10
num_well_detected: 6
num_redundant: 0
num_overmerged: 0
num_false_positive_units 3
num_bad: 3

**********
```

(continues on next page)

(continued from previous page)

```
rec1 klusta
             tp  fn fp num_gt num_tested tested_id
gt_unit_id
1             0  22  0     22          0        -1
2             0  26  0     26          0        -1
3             0  22  0     22          0        -1
4             0  25  0     25          0        -1
5             0  25  0     25          0        -1
6             0  27  0     27          0        -1
7             0  22  0     22          0        -1
8            13   9  0     22         13         4
9            24   4  0     28         24         2
10            0  22  0     22          0        -1
SUMMARY
-------
GT num_units: 10
TESTED num_units: 4
num_well_detected: 1
num_redundant: 0
num_overmerged: 1
num_false_positive_units 0
num_bad: 2

**********
rec0 klusta
             tp  fn fp num_gt num_tested tested_id
gt_unit_id
1            23   0  0     23         23         3
2             0  20  0     20          0        -1
3             0  24  0     24          0        -1
4             0  25  0     25          0        -1
5            18   6  8     24         26         2
6             0  23  0     23          0        -1
7             0  25  0     25          0        -1
8             0  20  0     20          0        -1
9             0  23  0     23          0        -1
10           21   1  0     22         21         4
SUMMARY
-------
GT num_units: 10
TESTED num_units: 4
num_well_detected: 2
num_redundant: 0
num_overmerged: 1
num_false_positive_units 0
num_bad: 1

**********
rec0 tridesclous
             tp  fn  fp num_gt num_tested tested_id
gt_unit_id
1            23   0   7     23         30         5
2             0  20   0     20          0        -1
3            24   0  24     24         48         4
4            25   0   0     25         25         0
5             0  24   0     24          0        -1
6            23   0   0     23         23         3
```

(continues on next page)

(continued from previous page)

```
7            25   0   0     25          25          1
8             0  20   0     20           0         -1
9             0  23   0     23           0         -1
10           22   0   0     22          22          2
SUMMARY
-------
GT num_units: 10
TESTED num_units: 6
num_well_detected: 4
num_redundant: 0
num_overmerged: 1
num_false_positive_units 0
num_bad: 0

**********
rec1 mountainsort4
             tp  fn fp num_gt num_tested tested_id
gt_unit_id
1            22   0  0     22          22          2
2             0  26  0     26           0         -1
3             0  22  0     22           0         -1
4            25   0  0     25          25          3
5             0  25  0     25           0         -1
6             0  27  0     27           0         -1
7            22   0  0     22          22          7
8             0  22  0     22           0         -1
9            28   0  5     28          33          9
10            0  22  0     22           0         -1
SUMMARY
-------
GT num_units: 10
TESTED num_units: 9
num_well_detected: 4
num_redundant: 0
num_overmerged: 2
num_false_positive_units 3
num_bad: 5

**********
rec1 tridesclous
             tp  fn  fp num_gt num_tested tested_id
gt_unit_id
1            22   0   0     22          22          2
2             0  26   0     26           0         -1
3             0  22   0     22           0         -1
4            25   0   0     25          25          1
5             0  25   0     25           0         -1
6             0  27   0     27           0         -1
7            22   0   0     22          22          0
8             0  22   0     22           0         -1
9            28   0  26     28          54          3
10            0  22   0     22           0         -1
SUMMARY
-------
GT num_units: 10
TESTED num_units: 5
num_well_detected: 3
```

(continues on next page)

(continued from previous page)

```
num_redundant: 0
num_overmerged: 2
num_false_positive_units 0
num_bad: 1
```

Collect synthetic dataframes and display

As shown previously, the performance is returned as a pandas dataframe. The `aggregate_performances_table` function, gathers all the outputs in the study folder and merges them in a single dataframe.

```
dataframes = study.aggregate_dataframes()
```

Pandas dataframes can be nicely displayed as tables in the notebook.

```
print(dataframes.keys())
```

Out:

```
dict_keys(['run_times', 'perf_by_units', 'count_units'])
```

```
print(dataframes['run_times'])
```

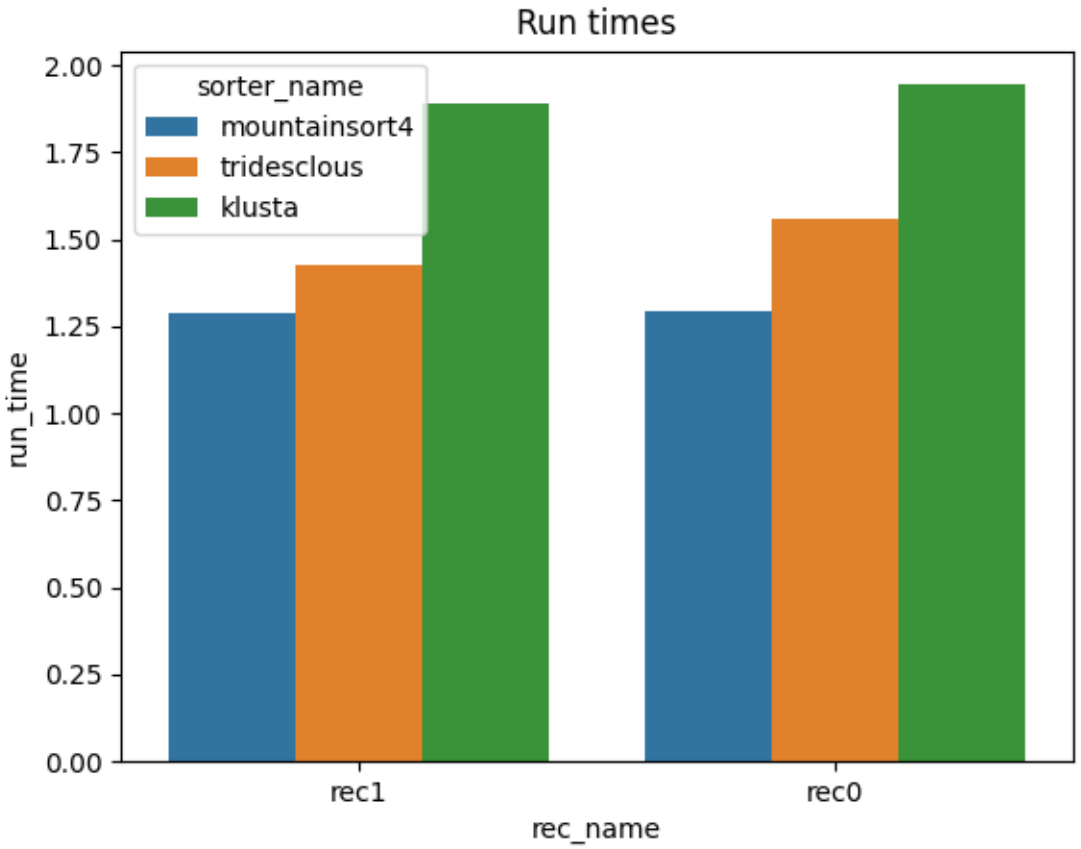
Out:

```
  rec_name    sorter_name  run_time
0     rec1  mountainsort4  1.287447
1     rec1    tridesclous  1.425234
2     rec0    tridesclous  1.560422
3     rec0         klusta  1.943773
4     rec1         klusta  1.888482
5     rec0  mountainsort4  1.293365
```

Easy plot with seaborn

Seaborn allows to easily plot pandas dataframes. Let's see some examples.

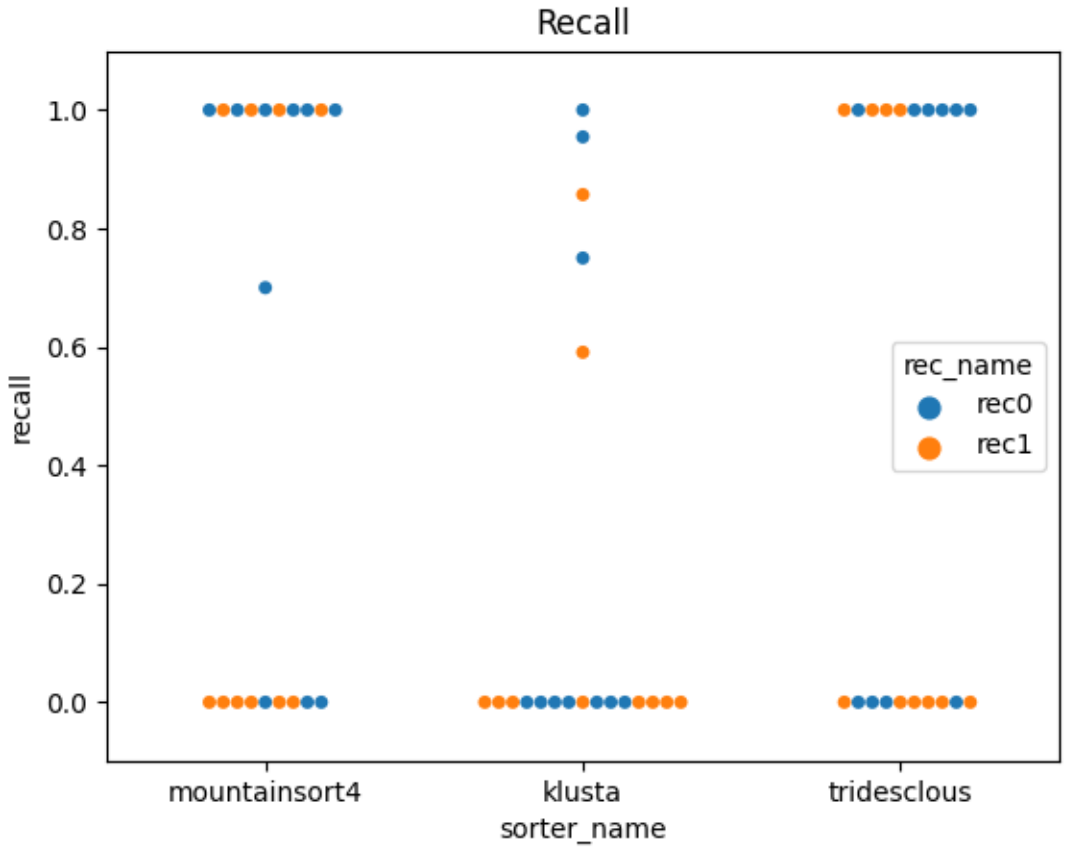
```
run_times = dataframes['run_times']
fig1, ax1 = plt.subplots()
sns.barplot(data=run_times, x='rec_name', y='run_time', hue='sorter_name', ax=ax1)
ax1.set_title('Run times')
```



Out:

```
Text(0.5, 1.0, 'Run times')
```

```
perfs = dataframes['perf_by_units']
fig2, ax2 = plt.subplots()
sns.swarmplot(data=perfs, x='sorter_name', y='recall', hue='rec_name', ax=ax2)
ax2.set_title('Recall')
ax2.set_ylim(-0.1, 1.1)
```



Out:

```
(-0.1, 1.1)
```

Total running time of the script: (0 minutes 11.485 seconds)

6.4.5 Explore sorters weaknesses with with ground-truth comparison

Here a syntetic dataset will demonstrate some weaknesses.

Standard weaknesses :

- not all units are detected
- a unit is detected, but not all of its spikes (false negatives)
- a unit is detected, but it detects too many spikes (false positives)

Other weaknesses:

- detect too many units (false positive units)
- detect units twice (or more) (reduntant units = oversplit units)
- several units are merged into one units (overmerged units)

To demonstarte this the script *generate_erroneous_sorting.py* generate a ground truth sorting with 10 units. We duplicate the results and modify it a bit to inject some “errors”:

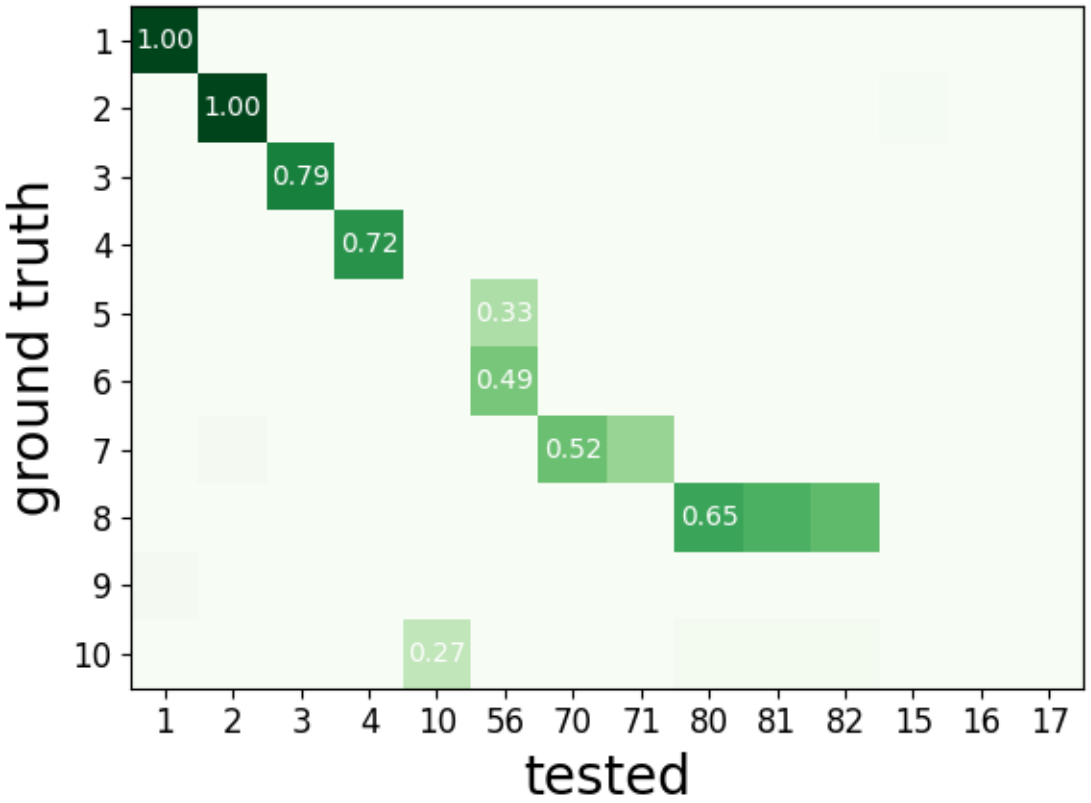
- unit 1 2 are perfect
- unit 3 4 have medium agreement
- unit 5 6 are over merge
- unit 7 is over split in 2 part
- unit 8 is redundant 3 times
- unit 9 is missing
- unit 10 have low agreement
- some units in tested do not exist at all in GT (15, 16, 17)

Import

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

import spikeinterface.extractors as se
import spikeinterface.sorters as ss
import spikeinterface.comparison as sc
import spikeinterface.widgets as sw

from generate_erroneous_sorting import generate_erroneous_sorting
```

Here the agreement matrix

```
sorting_true, sorting_err = generate_erroneous_sorting()
comp = sc.compare_sorter_to_ground_truth(sorting_true, sorting_err, exhaustive_
↪gt=True)
sw.plot_agreement_matrix(comp, ordered=False)
```

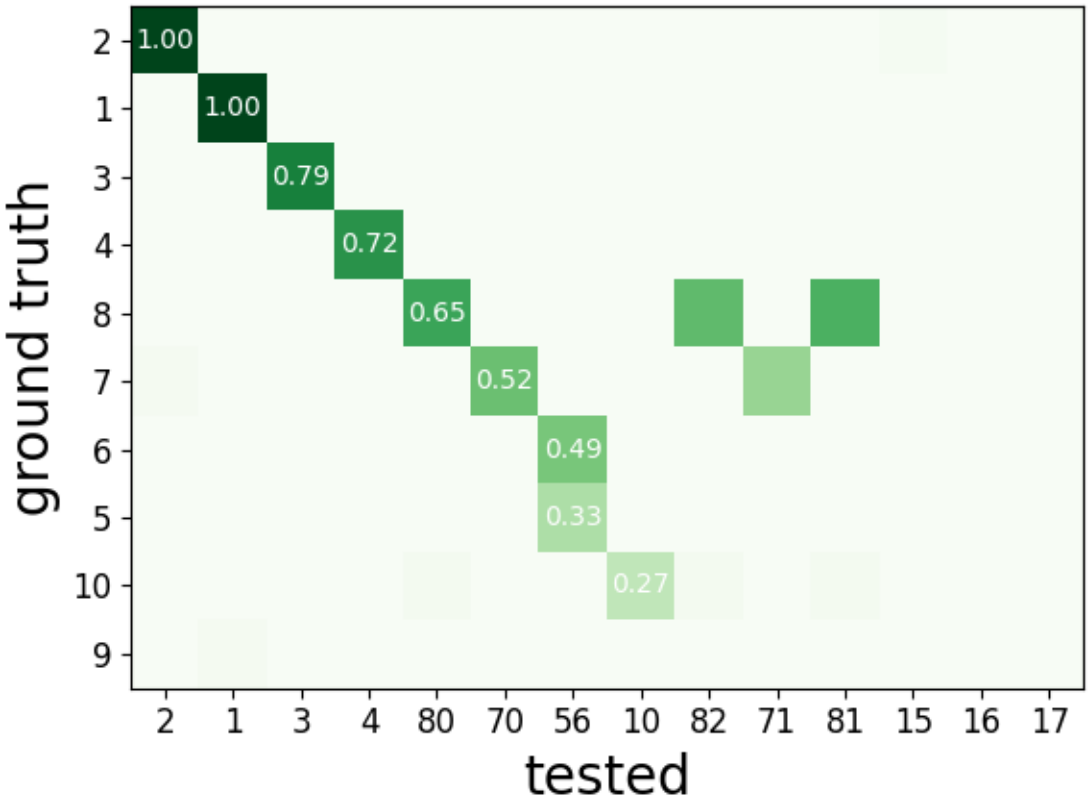


Out:

```
<spikewidgets.widgets.agreementmatrixwidget.agreementmatrixwidget.
↪AgreementMatrixWidget object at 0x7f3dfc25fbe0>
```

Here the same matrix but **ordered** It is now quite trivial to check that fake injected errors are enlighted here.

```
sw.plot_agreement_matrix(comp, ordered=True)
```



Out:

```
<spikewidgets.widgets.agreementmatrixwidget.agreementmatrixwidget.
↪AgreementMatrixWidget object at 0x7f3dfc701ac8>
```

Here we can see that only Units 1 2 and 3 are well detected with 'accuracy'>0.75

```
print('well_detected', comp.get_well_detected_units(well_detected_score=0.75))
```

Out:

```
well_detected [1, 2, 3]
```

Here we can explore **"false positive units"** units that do not exists in ground truth

```
print('false_positive', comp.get_false_positive_units(redundant_score=0.2))
```

Out:

```
false_positive [15, 16, 17]
```

Here we can explore **"redundant units"** units that do not exists in ground truth

```
print('redundant', comp.get_redundant_units(redundant_score=0.2))
```

Out:

```
redundant [71, 81, 82]
```

Here we can explore **"overmerged units"** units that do not exists in ground truth

```
print('overmerged', comp.get_overmerged_units(overmerged_score=0.2))
```

Out:

```
overmerged [56]
```

Here we can explore **"bad units"** units that a mixed a several possible errors.

```
print('bad', comp.get_bad_units())
```

Out:

```
bad [10, 56, 71, 81, 82, 15, 16, 17]
```

There is a convinient function to summary everything.

```
comp.print_summary(well_detected_score=0.75, redundant_score=0.2, overmerged_score=0.
↪2)



plt.show()
```

Out:

```
SUMMARY
-------
GT num_units: 10
TESTED num_units: 14
num_well_detected: 3
num_redundant: 3
num_overmerged: 1
num_false_positive_units 3
num_bad: 8
```

Total running time of the script: (0 minutes 0.519 seconds)

6.5 Widgets tutorials

The `widgets` module imports the spikewidgets package. It contains several plotting routines (widgets) for visualizing recordings and sorting data, probe layout, and many more!

6.5.1 RecordingExtractor Widgets Gallery

Here is a gallery of all the available widgets using RecordingExtractor objects.

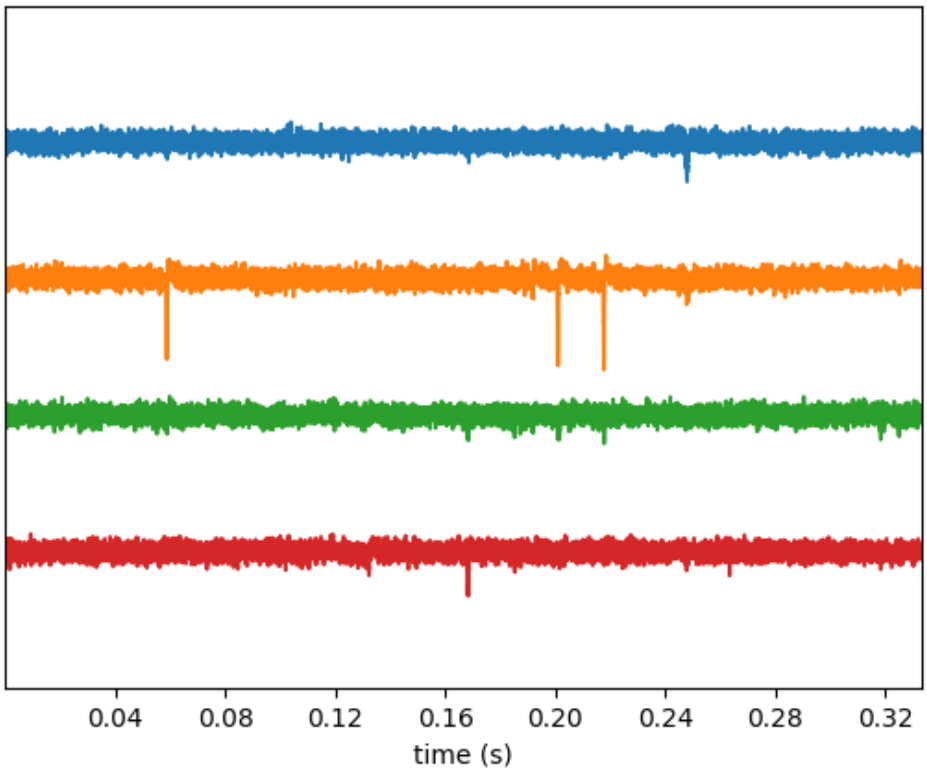
```
import spikeinterface.extractors as se
import spikeinterface.widgets as sw
```

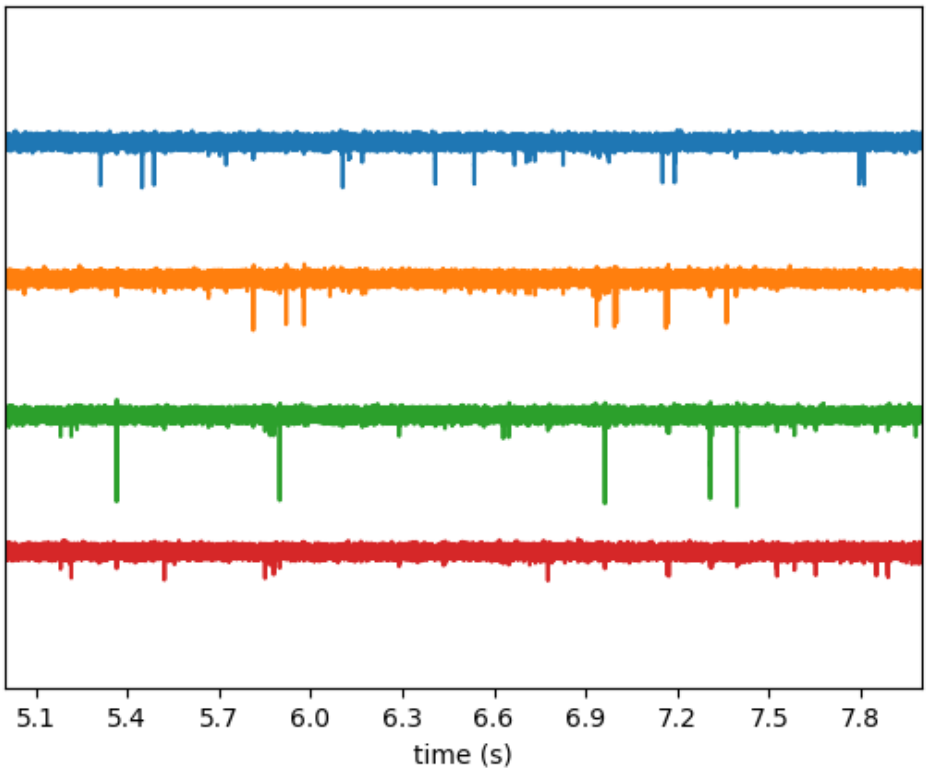
First, let's create a toy example with the *extractors* module:

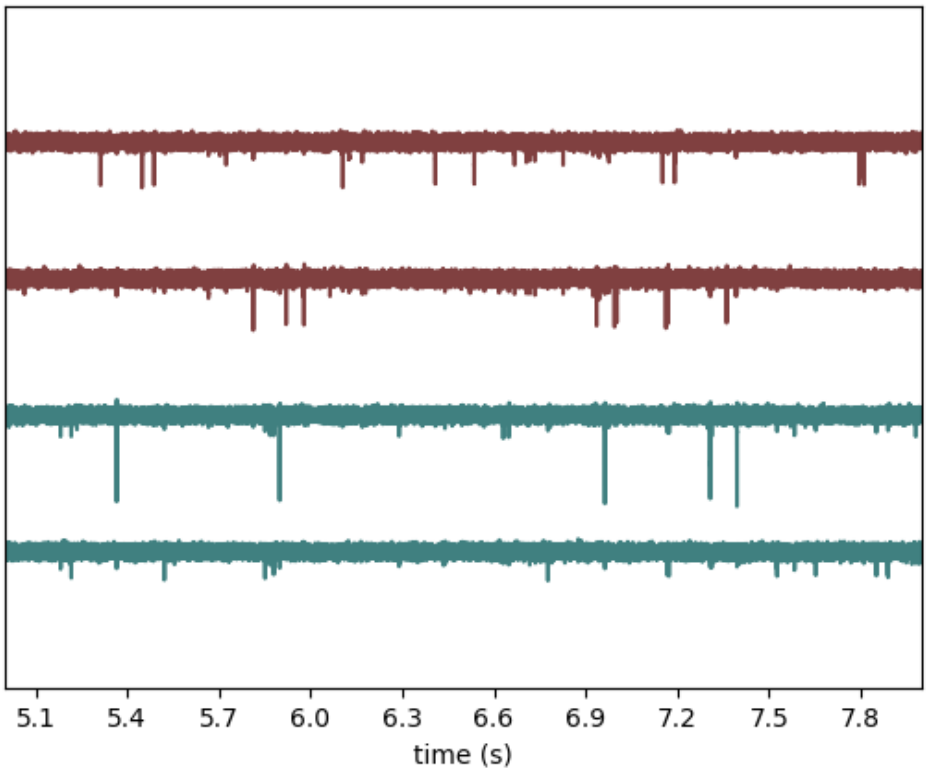
```
recording, sorting = se.example_datasets.toy_example(duration=10, num_channels=4,␣
↪seed=0)
```

plot_timeseries()

```
w_ts = sw.plot_timeseries(recording)

w_ts1 = sw.plot_timeseries(recording, trange=[5, 8])

recording.set_channel_groups(channel_ids=recording.get_channel_ids(), groups=[0, 0, 1,
↪ 1])
w_ts2 = sw.plot_timeseries(recording, trange=[5, 8], color_groups=True)
```




5.1
5.4
5.7
6.0
6.3
6.6
6.9
7.2
7.5
7.8
time (s)



-

Note: each function returns a widget object, which allows to access the figure and axis.

```
w_ts.figure.suptitle("Recording by group")
w_ts.ax.set_ylabel("Channel_ids")
```

Out:

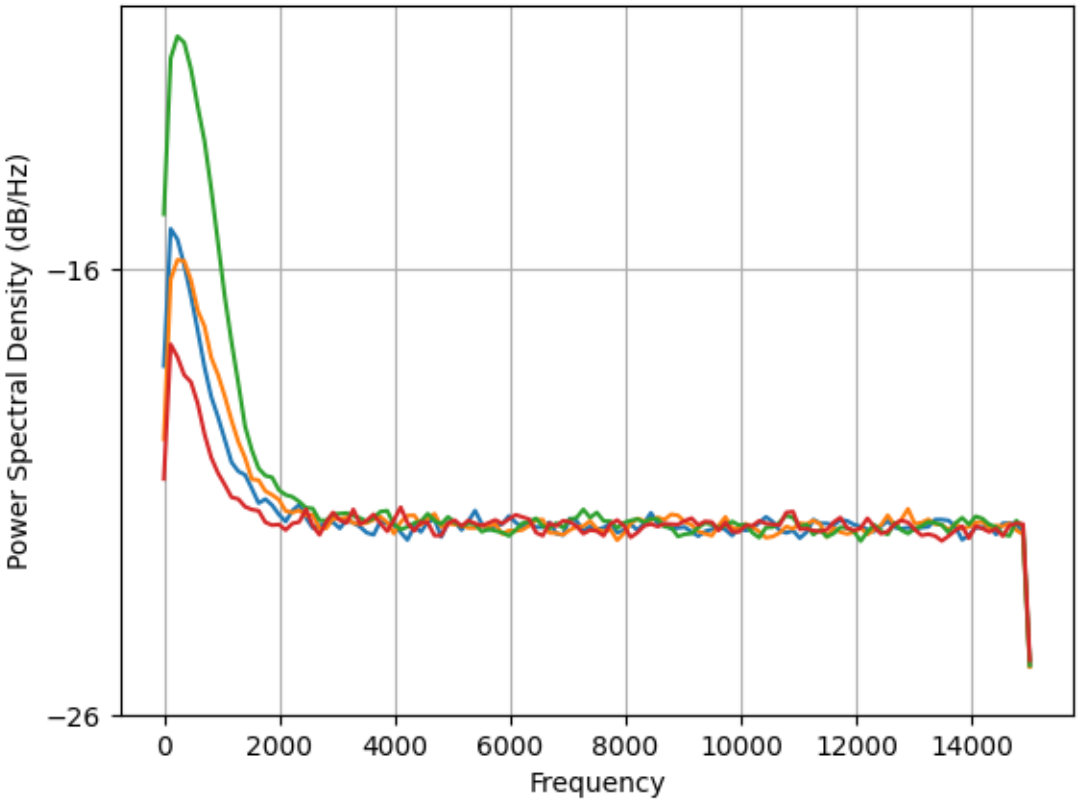
```
Text(74.44444444444444, 0.5, 'Channel_ids')
```

plot_electrode_geometry()

```
w_el = sw.plot_electrode_geometry(recording)
```

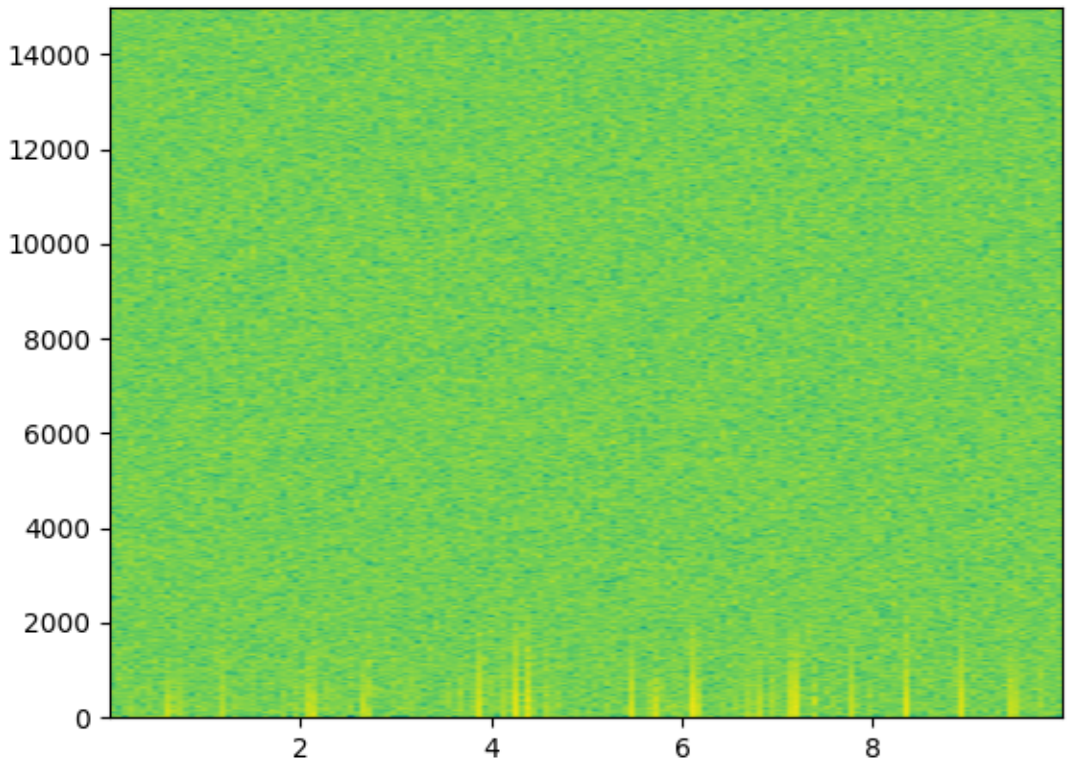
plot_spectrum()

```
w_sp = sw.plot_spectrum(recording)
```



plot_spectrogram()

```
w_spg = sw.plot_spectrogram(recording, channel=0, nfft=2048)
```



Total running time of the script: (0 minutes 0.943 seconds)

6.5.2 SortingExtractor Widgets Gallery

Here is a gallery of all the available widgets using SortingExtractor objects.

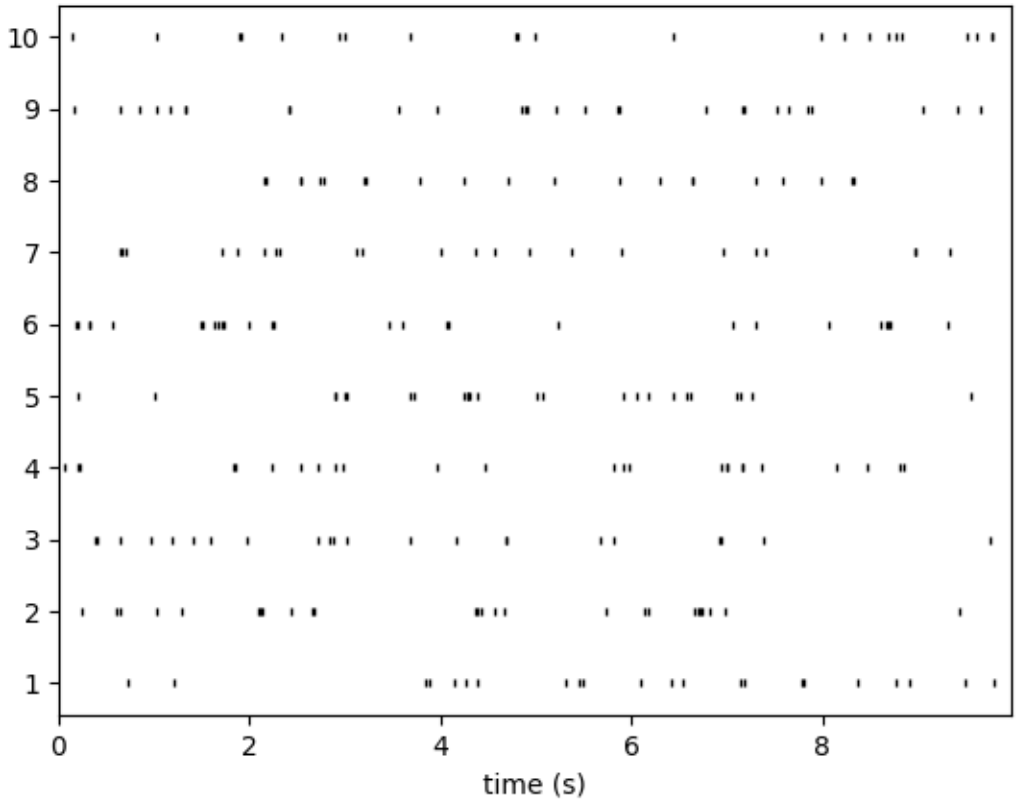
```
import spikeinterface.extractors as se
import spikeinterface.widgets as sw
```

First, let's create a toy example with the *extractors* module:

```
recording, sorting = se.example_datasets.toy_example(duration=10, num_channels=4,␣
↪seed=0)
```

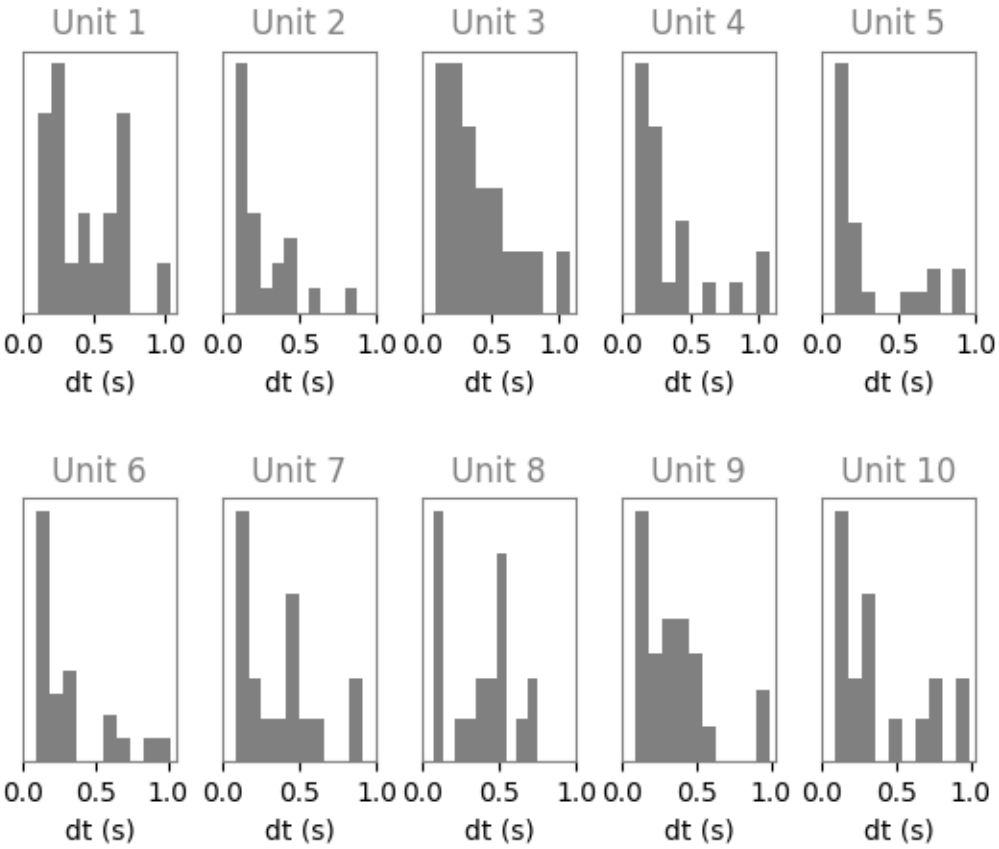
plot_rasters()

```
w_rs = sw.plot_rasters(sorting)
```



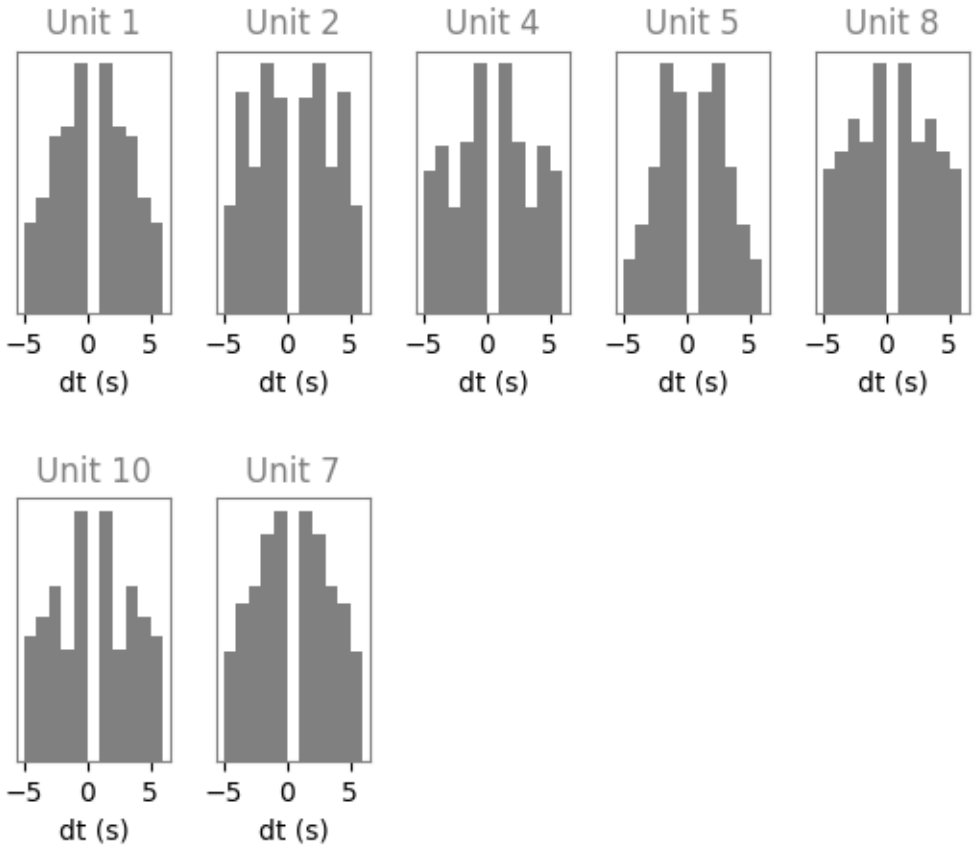
plot_isi_distribution()

```
w_isi = sw.plot_isi_distribution(sorting, bins=10, window=1)
```



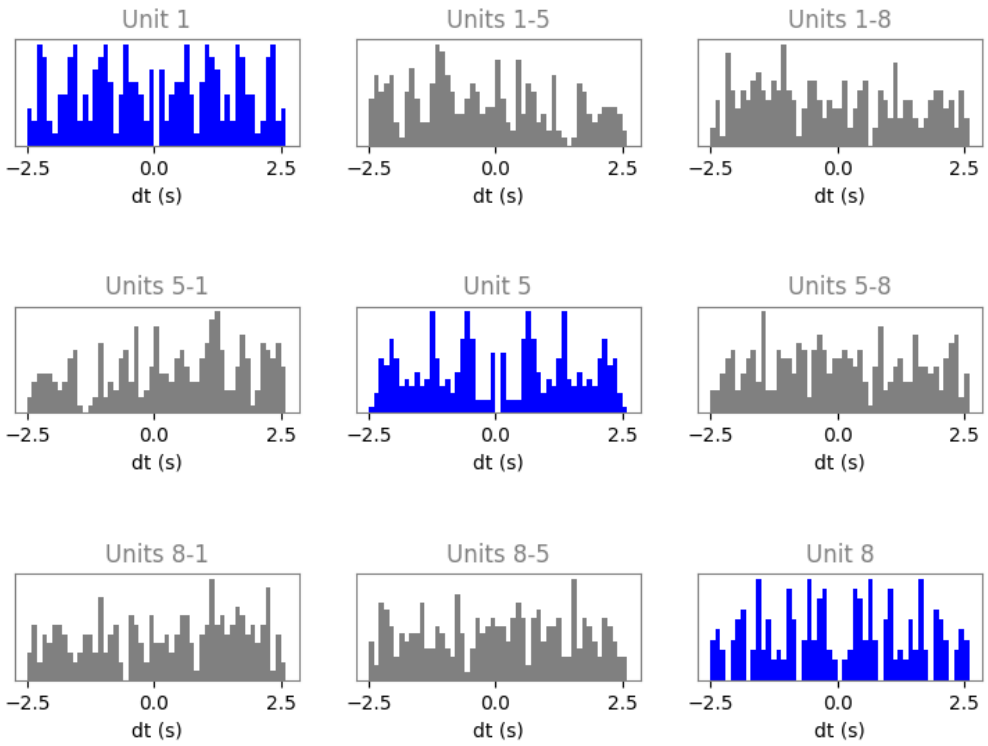
plot_autocorrelograms()

```
w_ach = sw.plot_autocorrelograms(sorting, bin_size=1, window=10, unit_ids=[1, 2, 4, 5,
↪ 8, 10, 7])
```



plot_crosscorrelograms()

```
w_cch = sw.plot_crosscorrelograms(sorting, unit_ids=[1, 5, 8], bin_size=0.1, window=5)
```



Total running time of the script: (0 minutes 1.440 seconds)

6.5.3 Recording+Sorting Widgets Gallery

Here is a gallery of all the available widgets using a pair of RecordingExtractor-SortingExtractor objects.

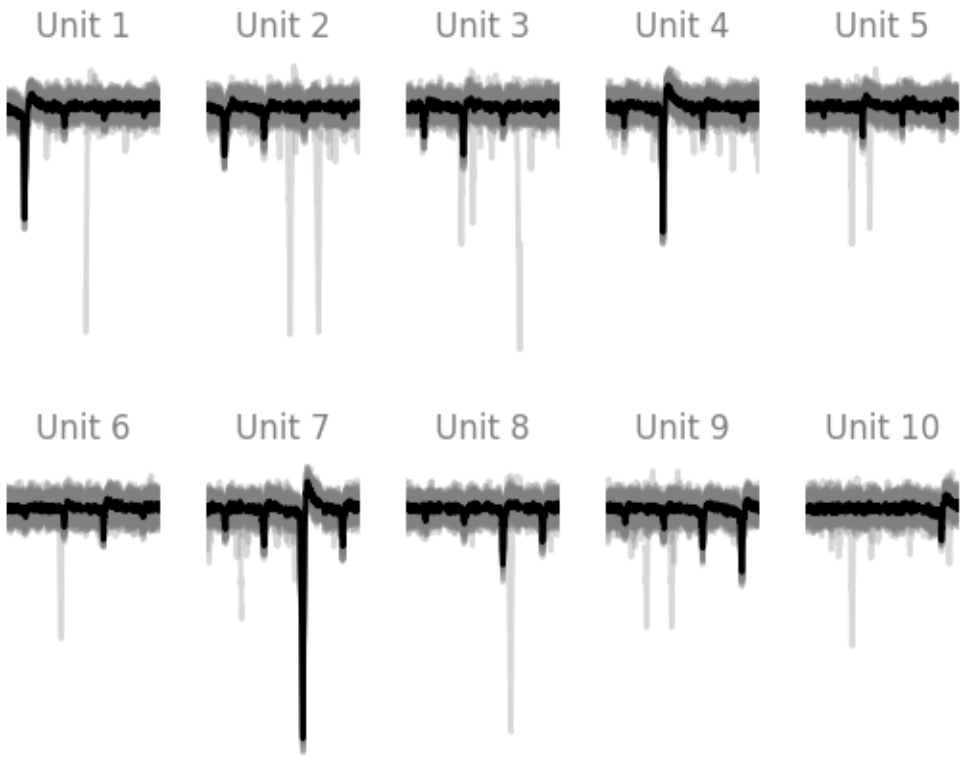
```
import spikeinterface.extractors as se
import spikeinterface.widgets as sw
```

First, let's create a toy example with the *extractors* module:

```
recording, sorting = se.example_datasets.toy_example(duration=10, num_channels=4,␣
↪seed=0)
```

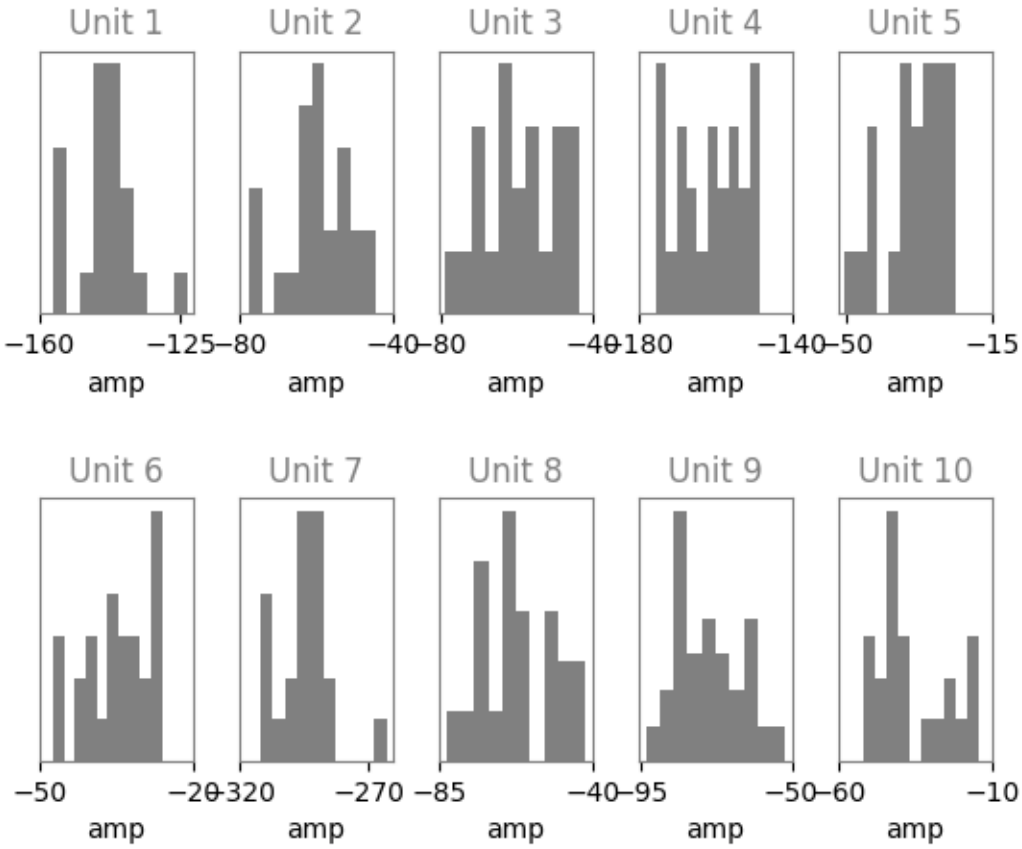
plot_unit_waveforms()

```
w_wf = sw.plot_unit_waveforms(recording, sorting, max_spikes_per_unit=100)
```



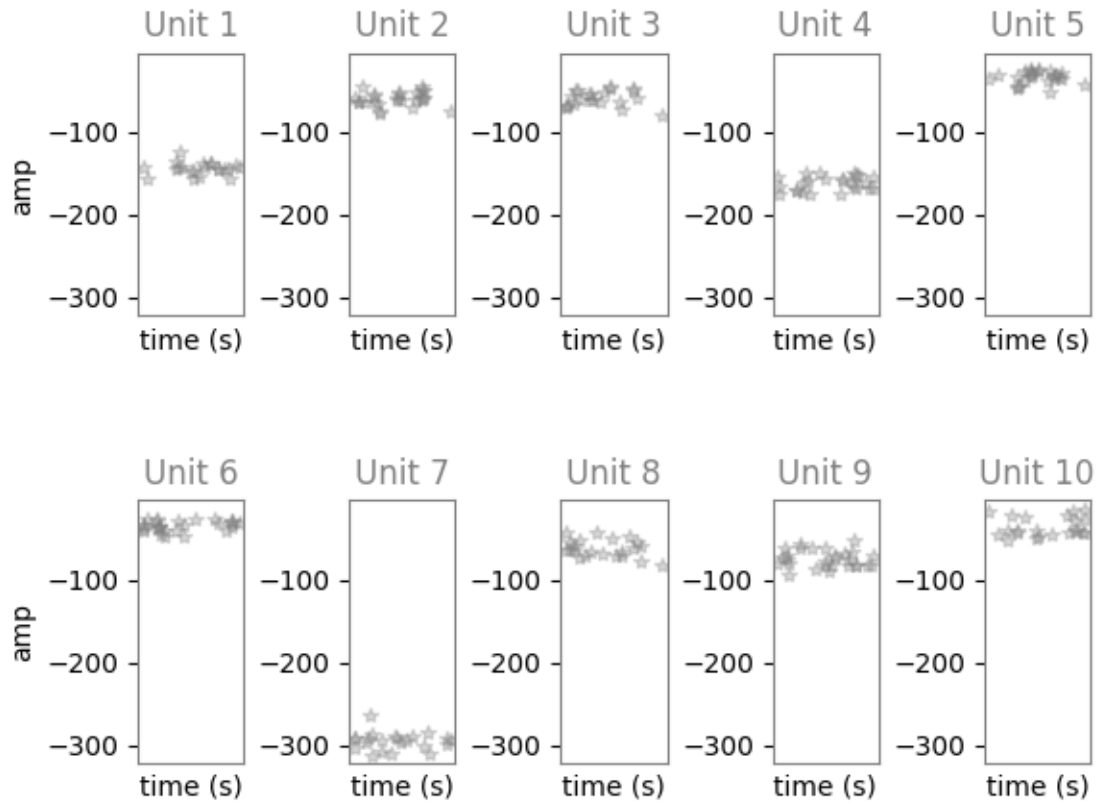
plot_amplitudes_distribution()

```
w_ampd = sw.plot_amplitudes_distribution(recording, sorting, max_spikes_per_unit=300)
```



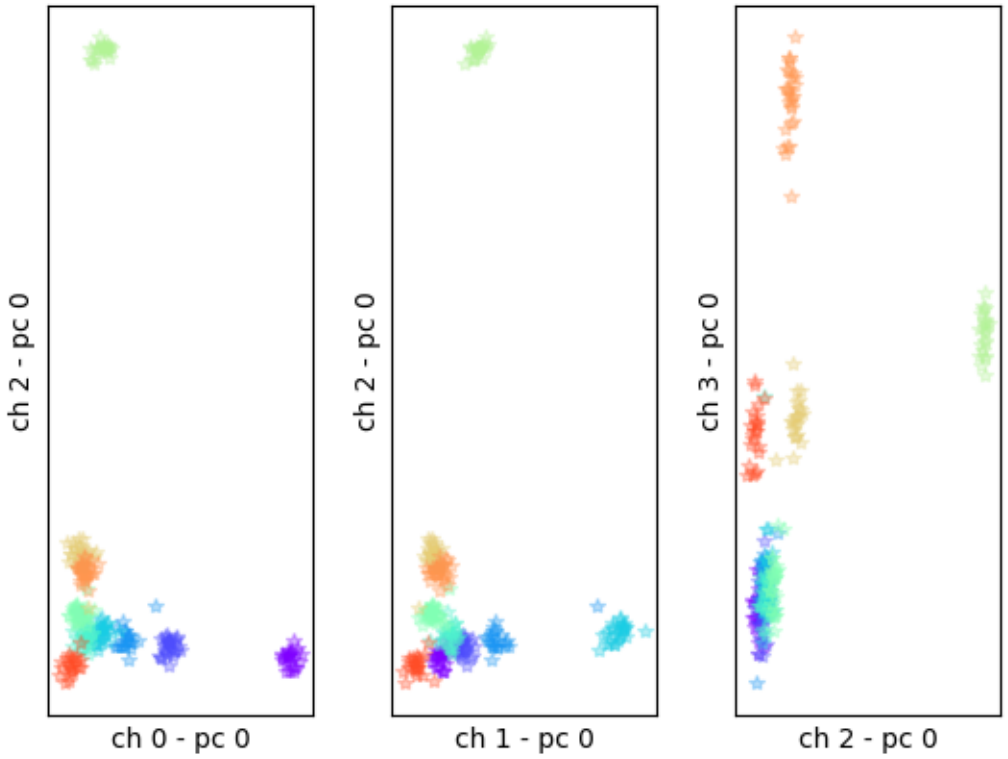
plot_amplitudes_timeseres()

```
w_ampt = sw.plot_amplitudes_timeseries(recording, sorting, max_spikes_per_unit=300)
```



plot_pca_features()

```
w_feat = sw.plot_pca_features(recording, sorting, colormap='rainbow', nproj=3, max_
↪spikes_per_unit=100)
```



Total running time of the script: (0 minutes 2.594 seconds)

6.5.4 Comparison Widgets Gallery

Here is a gallery of all the available widgets using SortingExtractor objects.

```
import spikeinterface.extractors as se
import spikeinterface.widgets as sw
```

First, let's create a toy example with the *extractors* module:

```
recording, sorting_true = se.example_datasets.toy_example(duration=10, num_channels=4,
↪ seed=0)
```

Let's run some spike sorting:

```
import spikeinterface.sorters as ss

sorting_MS4 = ss.run_mountainsort4(recording)
sorting_KL = ss.run_klusta(recording)
```

Out:

```
Warning! The recording is already filtered, but Mountainsort4 filter is enabled. You␣
↪can disable filters by setting 'filter' parameter to False
RUNNING SHELL SCRIPT: /home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/
↪checkouts/0.13.0/examples/modules/widgets/klusta_output/run_klusta.sh
/home/docs/checkouts/readthedocs.org/user_builds/spikeinterface/checkouts/0.13.0/doc/
↪sources/spikesorters/spikesorters/basesorter.py:158: ResourceWarning: unclosed file
↪<_io.TextIOWrapper name=63 encoding='UTF-8'>
  self._run(recording, self.output_folders[i])
```

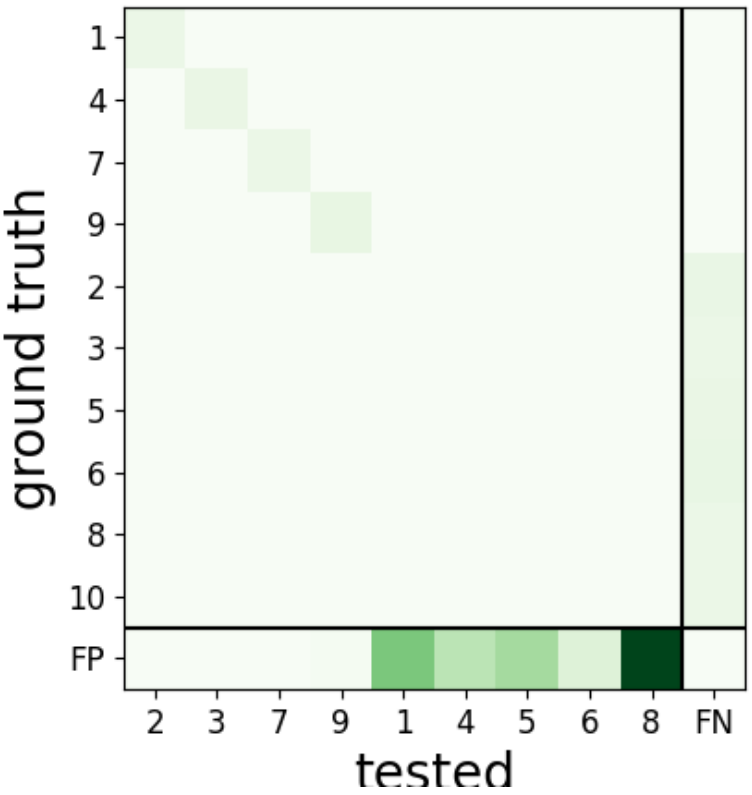
Widgets using SortingComparison

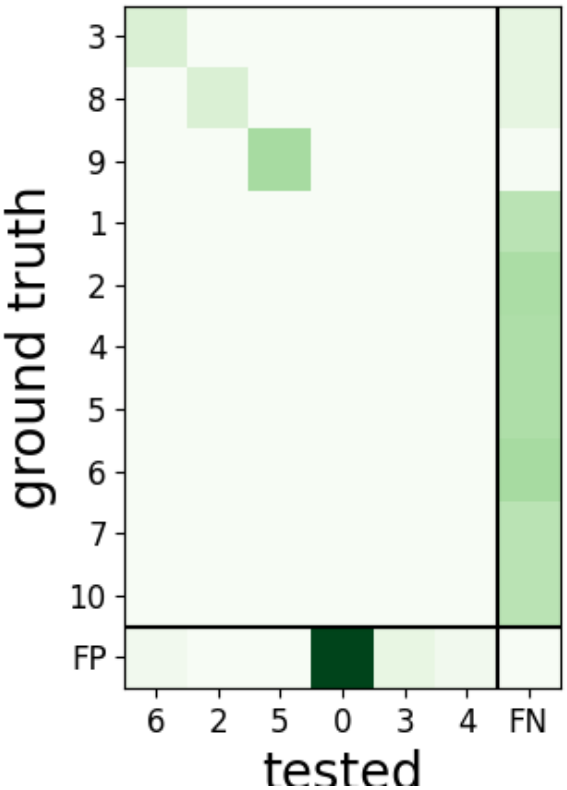
We can compare the spike sorting output to the ground-truth sorting `sorting_true` using the `comparison` module. `comp_MS4` and `comp_KL` are `SortingComparison` objects

```
import spikeinterface.comparison as sc

comp_MS4 = sc.compare_sorter_to_ground_truth(sorting_true, sorting_MS4)
comp_KL = sc.compare_sorter_to_ground_truth(sorting_true, sorting_KL)
```

plot_confusion_matrix()

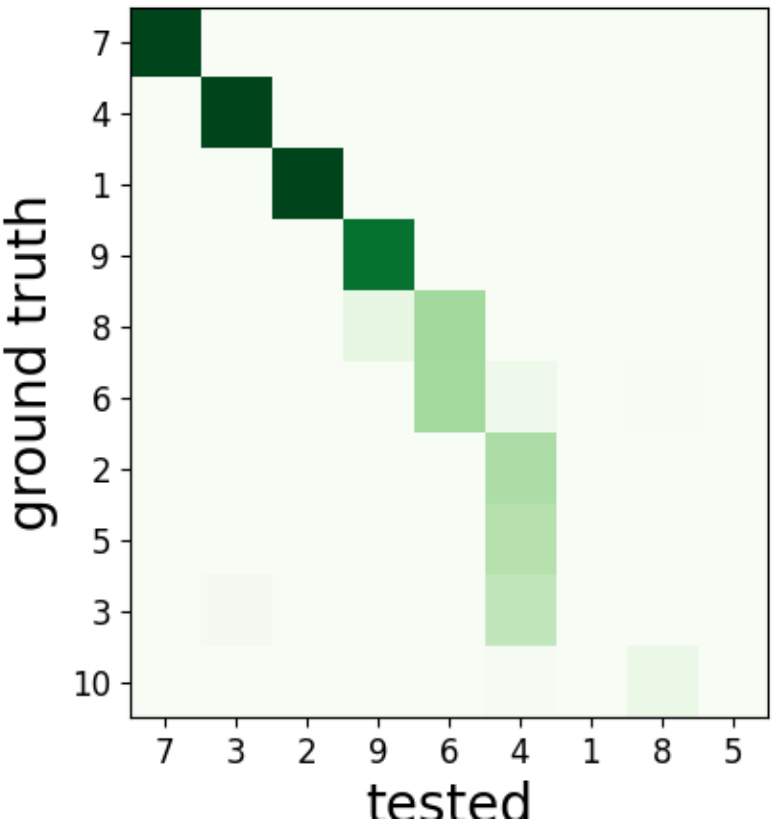
```
w_comp_MS4 = sw.plot_confusion_matrix(comp_MS4, count_text=False)
w_comp_KL = sw.plot_confusion_matrix(comp_KL, count_text=False)
```

- 



plot_agreement_matrix()

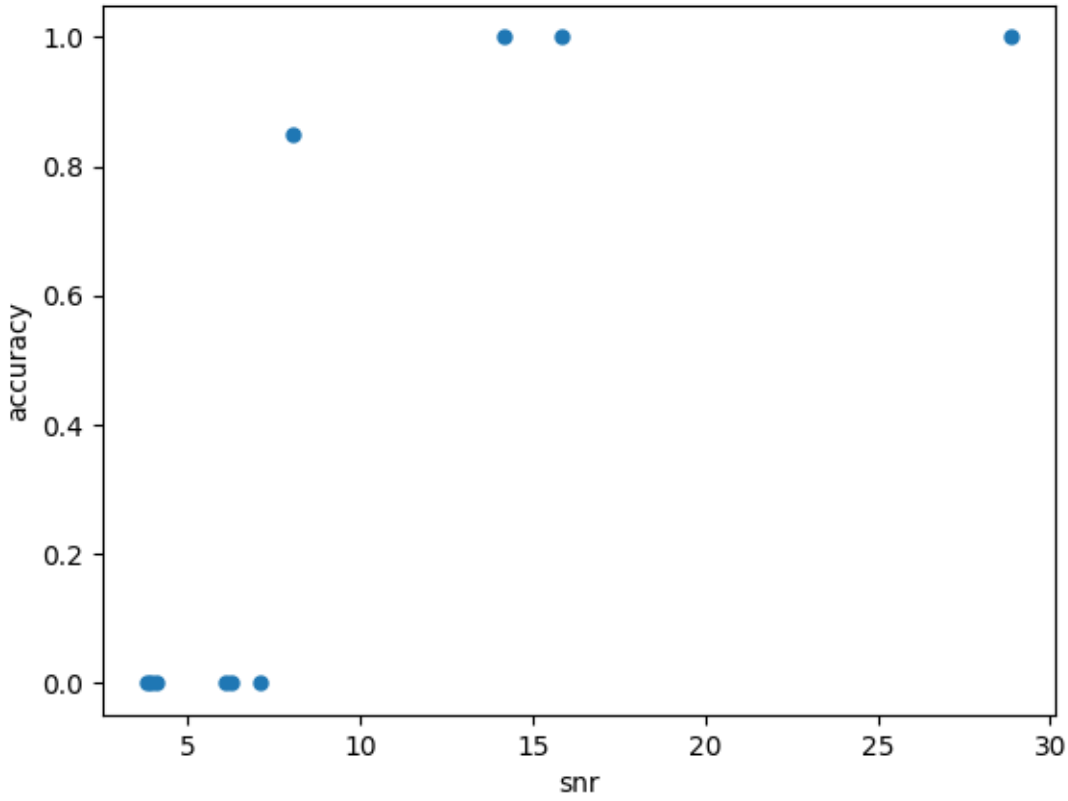
```
w_agr_MS4 = sw.plot_agreement_matrix(comp_MS4, count_text=False)
```



plot_sorting_performance()

We can also plot a performance metric (e.g. accuracy, recall, precision) with respect to a quality metric, for example signal-to-noise ratio. Quality metrics can be computed using the `toolkit.validation` submodule

```
import spikeinterface.toolkit as st

snrs = st.validation.compute_snrs(sorting_true, recording, save_as_property=True)

w_perf = sw.plot_sorting_performance(comp_MS4, property_name='snr', metric='accuracy')
```



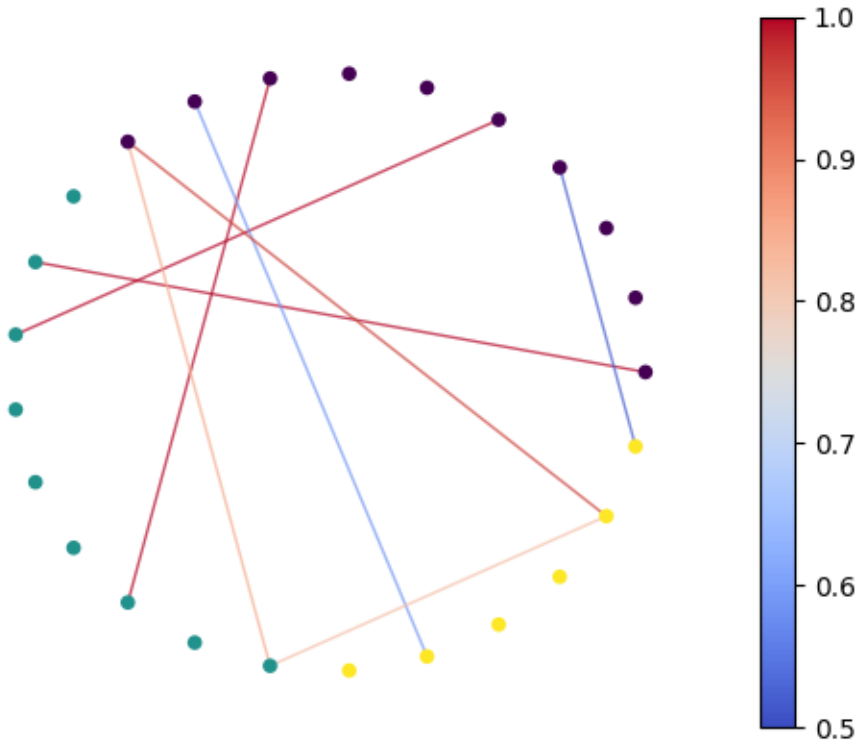
Widgets using MultiSortingComparison

We can also compare all three SortingExtractor objects, obtaining a `MultiSortingComparison` object.

```
multicomp = sc.compare_multiple_sorters([sorting_true, sorting_MS4, sorting_KL])
```

plot_multicomp_graph()

```
w_multi = sw.plot_multicomp_graph(multicomp, edge_cmap='coolwarm', node_cmap='viridis
↪', draw_labels=False,
                                  colorbar=True)
```



Total running time of the script: (0 minutes 4.355 seconds)

CHAPTER 7

Spike sorting comparison methods

SpikeInterface has a comparison module that can be used for three distinct use cases:

1. compare a spike sorting output with a ground-truth dataset
2. compare the output of two spike sorters (symmetric comparison)
3. compare the output of multiple spike sorters

Even if the three comparison cases share the same underlying idea (they compare spike trains!) the internal implementations are slightly different.

7.1 1. Comparison with ground truth

A ground-truth dataset can be a paired recording, in which the a neuron is recorded both extracellularly and with a patch or juxtacellular electrode (either **in vitro** or **in vivo**), or it can be a simulated dataset (**in silico**) using spiking activity simulators such as MEArec.

The comparison to ground-truth datasets is useful to benchmark spike sorting algorithms.

As an example, the SpikeForest platform benchmarks the performance of several spike sorters on a variety of available gorund-truth datasets on a daily basis. For more detailse see spikeforest notes.

This is the main workflow used to compute perfomance metrics:

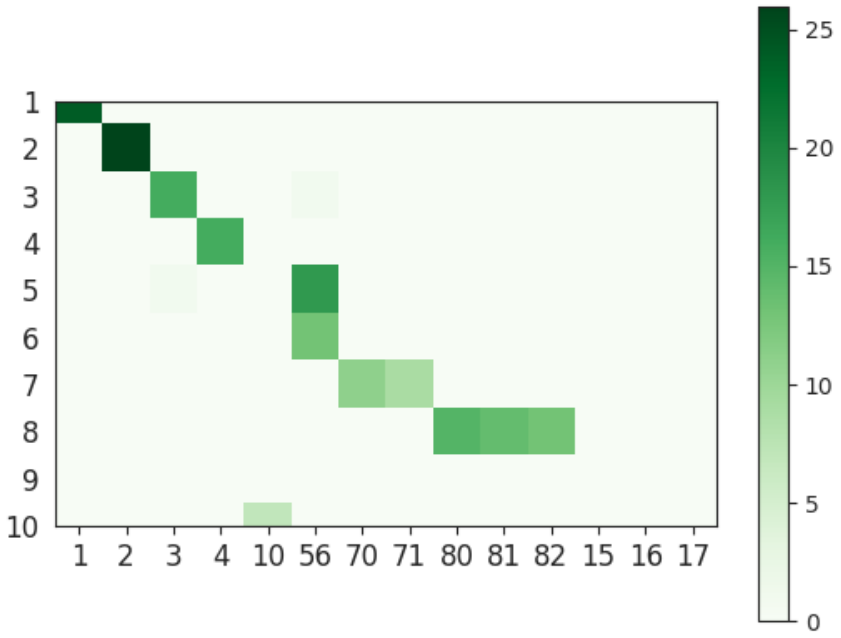
Given:

- **i = 1, . . . , n_gt** the list ground-truth (GT) units
- **k = 1, ..., n_tested** the list of tested units from spike sorting output
- **event_counts_GT[i]** the number of spikes for each units of GT unit
- **event_counts_ST[k]** the number of spikes for each units of tested unit

1. **Matching firing events**

For all pairs of GT unit and tested unit we first count how many events are matched within a *delta_time* tolerence (0.4 ms by defualt).

This gives a matrix called **match_event_count** of size *(n_gt X n_tested)*. This is an example of such matrices:



Note that this matrix represents the number of **true positive** (TP) spikes of each pair. We can also compute the number of **false negatives** (FN) and **false positive** (FP) spikes.

- **num_tp** [i, k] = match_event_count[i, k]
- **num_fn** [i, k] = event_counts_GT[i] - match_event_count[i, k]
- **num_fp** [i, k] = event_counts_ST[k] - match_event_count[i, k]

2. **Compute agreement score**

   Given the **match_event_count** we can then compute the **agreement_score**, which is normalized in the range [0, 1].

   This is done as follows:

   - agreement_score[i, k] = match_event_count[i, k] / (event_counts_GT[i] + event_counts_ST[k] - match_event_count[i, k])

   which is equivalent to:

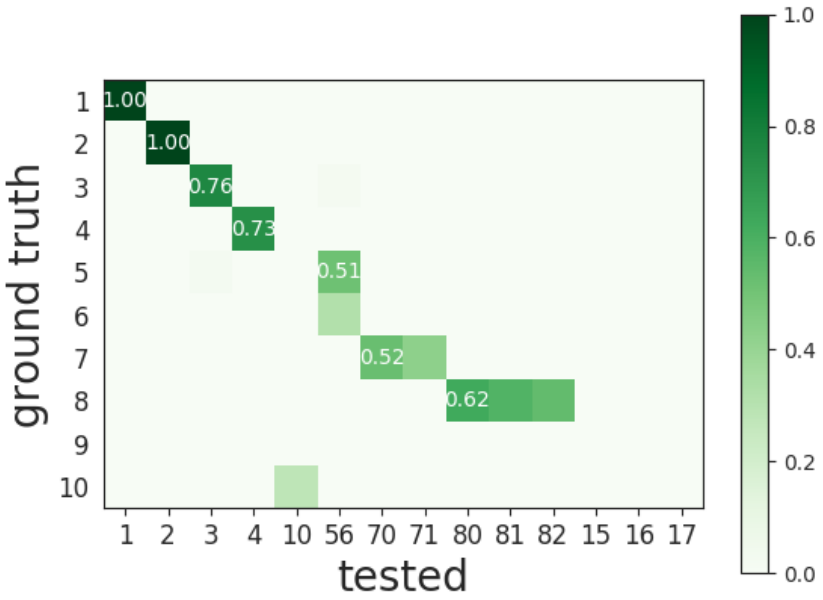
   - agreement_score[i, k] = num_tp[i, k] / (num_tp[i, k] + num_fp[i, k] + num_fn[i,k])

   or more practically:

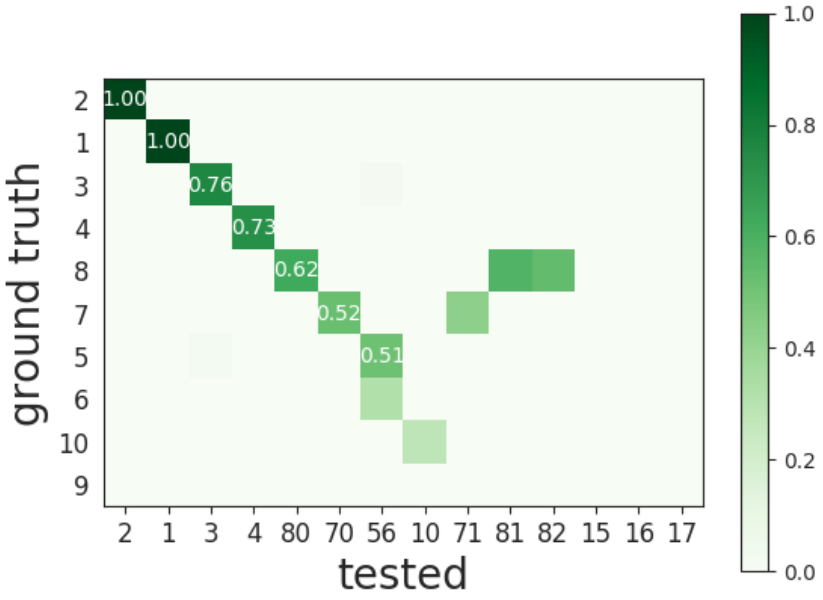
   - agreement_score[i, k] = intersection(I, K) / union(I, K)

   which is also equivalent to the **accuracy** metric.

   Here is an example of the agreement matrix, in which only scores > 0.5 are displayed:



This matrix can be ordered for a better visualization:



3. **Match units**

   During this step, given the **agreement_score** matrix each GT units can be matched to a tested units. For matching, a minimum **match_score** is used (0.5 by default). If the agreement is below this threshold, the possible match is discarded.

   There are two methods to perform the match: **hungarian** and **best** match.

The hungarian method finds the best association between GT and tested units. With this method, both GT and tested units can be matched only to another unit, or not matched at all.

For the **best** method, each GT unit is associated to a tested unit that has the **best** agreement_score, independently of all others units. Using this method several tested units can be associated to the same GT unit. Note that for the "best match" the minimum score is not the match_Score, but the **chance_score** (0.1 by default).

Here is an example of matching with the **hungarian** method. The first column represents the GT unit id and the second column the tested unit id. -1 means that the tested unit is not matched:

```
GT     TESTED
0      49
1      -1
2      26
3      44
4      -1
5      35
6      -1
7      -1
8      42
...
```

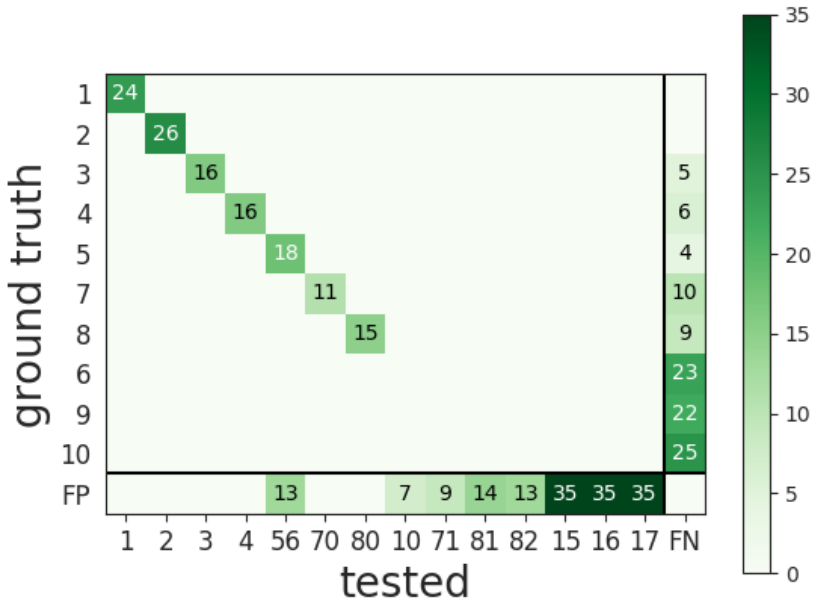
Note that the SpikeForest project uses the **best** match method.

4. **Compute performances**

   With the list of matched units we can compute performance metrics. Given : **tp** the number of true positive events, **fp** number of false positive event, **fn** the number of false negative event, **num_gt** the number of event of the matched tested units, the following metrics are computed for each GT unit:

   - accuracy = tp / (tp + fn + fp)
   - recall = tp / (tp + fn)
   - precision = tp / (tp + fp)
   - false_discovery_rate = fp / (tp + fp)
   - miss_rate = fn / num_gt

   The overall performances can be visualised with the **confusion matrix**, where the last columns counts **FN** and the last row counts **FP**.



7.1.1 More information about hungarian or best match methods

- **Hungarian**:

  Finds the best paring. If the matrix is square, then all units are associated. If the matrix is rectangular, then each row is matched. A GT unit (row) can be match one time only.

  - Pros
    - Each spike is counted only once
    - Hit score near chance levels are set to zero
    - Good FP estimation
  - Cons
    - Does not catch units that are split in several sub-units. Only the best math will be listed
    - More complicated implementation

- **Best**

  Each GT units is associated to the tested unit that has the best **agreement score**.

  - Pros:
    - Each GT unit is matched totally independently from others units
    - The accuracy score of a GT unit is totally independent from other units
    - It can identify over-merged units, as they would match multiple GT units
  - Cons:
    - A tested unit can be matched to multiple GT units, so some spikes can be counted several times
    - FP scores for units associated several times can be biased

    * Less robust with units having high firing rates

7.1.2 Classification of identified units

Tested units are classified depending on their performance. We identify three different classes:

- **well-detected** units
- **false positive** units
- **redundant** units
- **over-merged** units

A **well-detected** unit is a unit whose performance is good. By default, a good performance is measured by an accuracy greater than 0.8-

A **false positive** unit has low agreement scores for all GT units and it is not matched.

A **redundant** unit has a relatively high agreement (>= 0.2 by default), but it is not a best match. This means that it could either be an oversplit unit or a duplicate unit.

An **over-merged** unit has a relatively high agreement (>= 0.2 by default) for more than one GT unit.

7.2 2. Compare the output of two spike sorters (symmetric comparison)

The comparison of two sorter is a quite similar to the procedure of **compare to ground truth**. The difference is that no assumption is done on which is the units are ground-truth.

So the procedure is the following:

- **Matching firing events** : same a ground truth comparison
- **Compute agreement score** : same a ground truth comparison
- **Match units** : only with **hungarian** method

As there is no ground-truth information, performance metrics are not computed. However, the confusion and agreement matrices can be visualised to assess the level of agreement.

7.3 3. Compare the output of multiple spike sorters

Comparison of multiple sorters uses the following procedure:

1. Perform pairwise symmetric comparisons between spike sorters
2. Construct a graph in which nodes are units and edges are the agreements between units (of different sorters)
3. Extract units in agreement between two or more spike sorters
4. Build agreement spike trains, which only contain the spikes in agreement for the comparison with the highest agreement score

CHAPTER 8

Contribute

To contribute to SpikeInterface, a user/developer can help us integrate in a new recorded file format, a new sorted file format, or a new spike sorting algorithm.

8.1 Build a RecordingExtractor

Building a new `RecordingExtractor` for a specific file format is as simple as creating a new subclass based on the predefined base classes provided in the spikeextractors package.

To enable standardization among subclasses, the `RecordingExtractors` is an abstract base class which require a new subclass to **override all methods which are decorated with @abstractmethod**. The `RecordingExtractors` class has four abstract methods: `get_channel_ids()`, `get_num_frames()`, `get_sampling_frequency()`, and `get_traces()`. So all you need to do is create a class that inherits from `RecordingExtractor` and implements these four methods.

Along with these four methods, you can also optionally override the `write_recording()` function which enables any `RecordingExtractor` to be written into your format. Also, if you have an implementation of `get_snippets()` that is more efficient that the original implementation, you can optionally override that as well.

Any other methods, such as `set_channel_locations()` or `get_epoch()`, **should not** be overwritten as they are generic functions that any `RecordingExtractor` has access to upon initialization.

Finally, if your file format contains information about the channels (e.g. location, group, etc.), you are suggested to add that as a channel property upon initialization (this is optional).

An example of a RecordingExtractor that adds channel locations is shown here.

The contributed extractors are in the **spikeextractors/extractors** folder. You can fork the repo and create a new folder **myformatextractors** there. In the folder, create a new file named **myformatrecordingextractor.py**.

```
from spikeextractors import RecordingExtractor
from spikeextractors.extraction_tools import check_get_traces_args, check_get_ttl_args

try:
```

(continues on next page)

(continued from previous page)

```
    import mypackage
    HAVE_MYPACKAGE = True
except ImportError:
    HAVE_MYPACKAGE = False

class MyFormatRecordingExtractor(RecordingExtractor):
    """
    Description of your recording extractor

    Parameters
    ----------
    file_path: str or Path
        Path to myformat file
    extra_parameter: (type)
        What extra_parameter does
    """
    extractor_name = 'MyFormatRecording'
    has_default_locations = False  # set to True if extractor has default locations
    has_unscaled = False  # set to True if traces can be returned in raw format (e.g.␣
↪uint16/int16)
    installed = HAVE_MYPACKAGE  # check at class level if installed or not
    is_writable = True  # set to True if extractor implements `write_recording()`␣
↪function
    mode = 'file'  # 'file' if input is 'file_path', 'folder' if input 'folder_path',␣
↪'file_or_folder' if input is 'file_or_folder_path'
    installation_mesg = "To use the MyFormatRecordingExtractor install mypackage:␣
↪\n\n pip install mypackage\n\n"

    def __init__(self, file_path, extra_parameter):
        # check if installed
        assert self.installed, self.installation_mesg

        # instantiate base RecordingExtractor
        RecordingExtractor.__init__(self)

        ## All file specific initialization code can go here.

        # Important pieces of information include (if available): channel locations,␣
↪groups, gains, and offsets
        # To set these, one can use:
        # If the recording has default locations, they can be set as follows:
        self.set_channel_locations(locations)  # locations is a np.array (num_␣
↪channels x 2)
        # If the recording has intrinsic channel groups, they can be set as follows:
        self.set_channel_groups(groups)  # groups is a list or a np.array with length␣
↪num_channels
        # If the recording has unscaled traces, gains and offsets can be set as␣
↪follows:
        self.set_channel_gains(gains)  # gains is a list or a np.array with length␣
↪num_channels
        self.set_channel_offsets(gains)  # offsets is a list or a np.array with␣
↪length num_channels
        # If the recording has times in seconds that are not regularly sampled (e.g.␣
↪missing frames)
        # times in seconds can be set as follows:
        self.set_times(times) #
```

(continues on next page)

(continued from previous page)

```
        ### IMPORTANT ###
        #
        # gains and offsets are used to automatically convert raw data to uV (float)␣
↪in the following way:
        #
        # traces_uV = traces_raw * gains - offsets

    def get_channel_ids(self):

        # Fill code to get a list of channel_ids. If channel ids are not specified,␣
↪you can use:
        # channel_ids = range(num_channels)

        return channel_ids

    def get_num_frames(self):

        # Fill code to get the number of frames (samples) in the recordings.

        return num_frames

    def get_sampling_frequency(self, unit_id, start_frame=None, end_frame=None):

        # Fill code to get the sampling frequency of the recordings.

        return sampling_frequency

    @check_get_traces_args
    def get_traces(self, channel_ids=None, start_frame=None, end_frame=None, return_
↪scaled=True):
        '''This function extracts and returns a trace from the recorded data from the
        given channels ids and the given start and end frame. It will return
        traces from within three ranges:

            [start_frame, t_start+1, ..., end_frame-1]
            [start_frame, start_frame+1, ..., final_recording_frame - 1]
            [0, 1, ..., end_frame-1]
            [0, 1, ..., final_recording_frame - 1]

        if both start_frame and end_frame are given, if only start_frame is
        given, if only end_frame is given, or if neither start_frame or end_frame
        are given, respectively. Traces are returned in a 2D array that
        contains all of the traces from each channel with dimensions
        (num_channels x num_frames). In this implementation, start_frame is inclusive
        and end_frame is exclusive conforming to numpy standards.

        Parameters
        ----------
        start_frame: int
            The starting frame of the trace to be returned (inclusive).
        end_frame: int
            The ending frame of the trace to be returned (exclusive).
        channel_ids: array_like
            A list or 1D array of channel ids (ints) from which each trace will be
            extracted.
        return_scaled: bool
            If True, traces are returned after scaling (using gain/offset). If False,␣
↪the raw traces are returned
```

(continues on next page)

(continued from previous page)

```
        Returns
        ----------
        traces: numpy.ndarray
            A 2D array that contains all of the traces from each channel.
            Dimensions are: (num_channels x num_frames)
        '''

        # Fill code to get the the traces of the specified channel_ids, from start_
↪frame to end_frame
        #
        ### IMPORTANT ###
        #
        # If raw traces are available (e.g. int16/uint16), this function should␣
↪return the raw traces only!
        # If gains and offsets are set in the init, the conversion to float is done␣
↪automatically (depending on the
        # return_scaled) argument.

        return traces

    # optional
    @check_get_ttl_args
    def get_ttl_events(self, start_frame=None, end_frame=None, channel_id=0):
        '''
        Returns an array with frames of TTL signals. To be implemented in sub-classes

        Parameters
        ----------
        start_frame: int
            The starting frame of the ttl to be returned (inclusive)
        end_frame: int
            The ending frame of the ttl to be returned (exclusive)
        channel_id: int
            The TTL channel id

        Returns
        -------
        ttl_frames: array-like
            Frames of TTL signal for the specified channel
        ttl_state: array-like
            State of the transition: 1 - rising, -1 - falling
        '''

        # Fill code to return ttl frames and states

        return ttl_frames, ttl_states

    .
    .
    .
    .
    . #Optional functions and pre-implemented functions that a new RecordingExtractor␣
↪doesn't need to implement
    .
    .
    .
```

(continues on next page)

(continued from previous page)

```
    .

    @staticmethod
    def write_recording(recording, save_path, other_params):
        '''
        This is an example of a function that is not abstract so it is optional if␣
↪you want to override it.
        It allows other RecordingExtractor to use your new RecordingExtractor to␣
↪convert their recorded data into
        your recording file format.
        '''
```

When you are done you should add your `RecordingExtractor` to the **extarctorlist.py** file. You can optionally write a test in the **tests/test_extractors.py** (this is easier if a `write_recording` function is implemented).

Finally, make a pull request to the spikeextractor repo, so we can review the code and merge it to the spikeextractors!

8.2 Build a SortingExtractor

Building a new `SortingExtractor` for a specific file format is as simple as creating a new subclass based on the predefined base classes provided in the spikeextractors package.

To enable standardization among subclasses, the `SortingExtractor` is an abstract base class which require a new subclass to **override all methods which are decorated with @abstractmethod**. The `SortingExtractor` class has two abstract methods: `get_unit_ids()`, `get_unit_spike_trains()`. So all you need to do is create a class that inherits from `:code:`SortingExtractor`` and implements these two methods.

Along with these two methods, you can also optionally override the `write_sorting()` function which enables any `SortingExtractor` to be written into your format.

Any other methods, such as `set_unit_spike_features()` or `clear_unit_property()`, **should not** be overwritten as they are generic functions that any `SortingExtractor` has access to upon initialization.

Finally, if your file format contains information about the units (e.g. location, morphology, etc.) or spikes (e.g. locations, pcs, etc.), you are suggested to add that as either unit properties or spike features upon initialization (this is optional).

The contributed extractors are in the **spikeextractors/extractors** folder. You can fork the repo and create a new folder **myformatextractors** there. In the folder, create a new file named **myformatsortingextractor.py**.

```
from spikeextractors import SortingExtractor
from spikeextractors.extraction_tools import check_get_unit_spike_train

 try:
    import mypackage
    HAVE_MYPACKAGE = True
except ImportError:
    HAVE_MYPACKAGE = False

class MyFormatSortingExtractor(SortingExtractor):
    """
    Description of your sorting extractor

    Parameters
    ----------
    file_path: str or Path
```

(continues on next page)

(continued from previous page)

```
        Path to myformat file
    extra_parameter_1: (type)
        What extra_parameter_1 does
    extra_parameter_2: (type)
        What extra_parameter_2 does
    """
    extractor_name = 'MyFormatSorting'
    installed = HAVE_MYPACKAGE  # check at class level if installed or not
    is_writable = True # set to True if extractor implements `write_sorting()`␣
↪function
    mode = 'file'  # 'file' if input is 'file_path', 'folder' if input 'folder_path',
↪'file_or_folder' if input is 'file_or_folder_path'
    installation_mesg = "To use the MyFormatSortingExtractor extractors, install␣
↪mypackage: \n\n pip install mypackage\n\n"

    def __init__(self, file_path, extra_parameter_1, extra_parameter_2):
        # check if installed
        assert self.installed, self.installation_mesg

        # instantiate base SortingExtractor
        SortingExtractor.__init__(self)

        ## All file specific initialization code can go here.
        # If your format stores the sampling frequency, you can overwrite the self._
↪sampling_frequency. This way,
        # the base method self.get_sampling_frequency() will return the correct␣
↪sampling frequency

        self._sampling_frequency = my_sampling_frequency

    def get_unit_ids(self):

        #Fill code to get a unit_ids list containing all the ids (ints) of detected␣
↪units in the recording

        return unit_ids

    @check_get_unit_spike_train
    def get_unit_spike_train(self, unit_id, start_frame=None, end_frame=None):

        '''Code to extract spike frames from the specified unit.
        It will return spike frames from within three ranges:
            [start_frame, t_start+1, ..., end_frame-1]
            [start_frame, start_frame+1, ..., final_unit_spike_frame - 1]
            [0, 1, ..., end_frame-1]
            [0, 1, ..., final_unit_spike_frame - 1]
        if both start_frame and end_frame are given, if only start_frame is
        given, if only end_frame is given, or if neither start_frame or end_frame
        are given, respectively. Spike frames are returned in the form of an
        array_like of spike frames. In this implementation, start_frame is inclusive
        and end_frame is exclusive conforming to numpy standards.

        '''

        return spike_train

    .
```

(continues on next page)

(continued from previous page)

```
    .
    .
    .
    . #Optional functions and pre-implemented functions that a new SortingExtractor␣
↪doesn't need to implement
    .
    .
    .
    .

    @staticmethod
    def write_sorting(sorting, save_path):
        '''
        This is an example of a function that is not abstract so it is optional if␣
↪you want to override it. It allows other
        SortingExtractors to use your new SortingExtractor to convert their sorted␣
↪data into your
        sorting file format.
        '''
```

When you are done you can optionally write a test in the **tests/test_extractors.py** (this is easier if a `write_sorting` function is implemented).

Finally, make a pull request to the spikeextractors repo, so we can review the code and merge it to the spikeextractors!

8.3 Implement a spike sorter

Implementing a new spike sorter for a specific file format is as simple as creating a new subclass based on the predefined base classes provided in the spikesorters package.

To enable standardization among subclasses, the `BaseSorter` is base class which require a new subclass to override a few methods.

The contributed extractors are in the **spikesorters** folder. You can fork the repo and create a new folder **myspikesorter** there. In the folder, create a new file named **myspikesorter.py**. Additional configuration files must be placed in the same folder.

You can start by importing the base class:

```
import spikeextractors as se
from ..basesorter import BaseSorter
```

In order to check if your spike sorter is installed, a `try - except` block is used. For example, if your sorter is implemented in Python (installed with the package `myspikesorter`), this block will look as follows:

```
try:
    import myspikesorter
    HAVE_MSS = True
except ImportError:
    HAVE_MSS = False
```

Then, you can start creating a new class:

```
class MySpikeSorter(BaseSorter):
"""
```

(continues on next page)

(continued from previous page)

```
Brief description (optional)
"""

sorter_name = 'myspikesorter'
installed = HAVE_MSS

_default_params = {
    'param1': None,
    'param2': 2,
    }

_params_description = {
    'param1': 'Description for param1',
    'param1': 'Description for param1',
}

installation_mesg = """
    >>> pip install myspikesorter
    More information on MySpikesorter at:
        https://myspikesorterwebsite.com
"""
```

Now you can start filling out the required methods:

```
def __init__(self, **kargs):
    BaseSorter.__init__(self, **kargs)

# optional
@staticmethod
def get_sorter_version():
    return myspikesorter.__version__

@classmethod
def is_installed(cls):

    # Fill code to check sorter installation. It returns a boolean
    return HAVE_MSS

def _setup_recording(self, recording, output_folder):

    # Fill code to set up the recording: convert to required file, parse config files,
↪ etc.
    # The files should be placed in the 'output_folder'

def _run(self, recording, output_folder):

    # Fill code to run your spike sorter based on the files created in the _setup_
↪recording()
    # You can run CLI commands (e.g. klusta, spykingcircus, tridescous), pure Python␣
↪code (e.g. Mountainsort4,
    # Herding Spikes), or even MATLAB code (e.g. Kilosort, Kilosort2, Ironclust)

@staticmethod
def get_result_from_folder(output_folder):

    # If your spike sorter has a specific file format, you should implement a␣
↪SortingExtractor in spikeextractors.
```

(continues on next page)

(continued from previous page)

```
    # Let's assume you have done so, and the extractor is called␣
↪MySpikeSorterSortingExtractor

    sorting = se.MySpikeSorterSortingExtractor(output_folder)
    return sorting
```

When your spike sorter class is implemented, you have to add it to the list of available spike sorters in the sorterlist.py. Moreover, you have to add a launcher function:

```
def run_myspikesorter(*args, **kargs):
    return run_sorter('myspikesorter', *args, **kargs)
```

When you are done you can optionally write a test in **tests/test_myspikesorter.py**. In order to be tested, you can install the required packages by changing the **.travis.yml**. Note that MATLAB based tests cannot be run at the moment, but we recommend testing the implementation locally.

Finally, make a pull request to the spikesorters repo, so we can review the code and merge it to the spikesorters!

CHAPTER 9

API

9.1 Module spikeinterface.extractors

class spikeextractors.baseextractor.**BaseExtractor**

add_epoch(*epoch_name*, *start_frame*, *end_frame*)
This function adds an epoch to your extractor that tracks a certain time period. It is stored in an internal dictionary of start and end frame tuples.

epoch_name: str The name of the epoch to be added

start_frame: int The start frame of the epoch to be added (inclusive)

end_frame: int The end frame of the epoch to be added (exclusive). If set to None, it will include the entire sorting after the start_frame

allocate_array(*memmap*, *shape=None*, *dtype=None*, *name=None*, *array=None*)
Allocates a memory or memmap array

memmap: bool If True, a memmap array is created in the sorting temporary folder

shape: tuple Shape of the array. If None array must be given

dtype: dtype Dtype of the array. If None array must be given

name: str or None Name (root) of the file (if memmap is True). If None, a random name is generated

array: np.array If array is given, shape and dtype are initialized based on the array. If memmap is True, the array is then deleted to clear memory

arr: np.array or np.memmap The allocated memory or memmap array

annotate(*annotation_key*, *value*, *overwrite=False*)
This function adds an entry to the annotations dictionary.

annotation_key: str An annotation stored by the Extractor

value: The data associated with the given property name. Could be many formats as specified by the user

overwrite: bool If True and the annotation already exists, it is overwritten

copy_annotations(*extractor*)
Copy object properties from another extractor to the current extractor.

extractor: Extractor The extractor from which the annotations will be copied

copy_epochs(*extractor*)
Copy epochs from another extractor.

extractor: BaseExtractor The extractor from which the epochs will be copied

del_memmap_file(*memmap_file*)
Safely deletes instantiated memmap file.

memmap_file: str or Path The memmap file to delete

dump_to_dict(*relative_to=None*)
Dumps recording to a dictionary. The dictionary be used to re-initialize an extractor with spikeextractors.load_extractor_from_dict(dump_dict)

relative_to: str, Path, or None If not None, file_paths are serialized relative to this path

dump_dict: dict Serialized dictionary

dump_to_json(*file_path=None*, *relative_to=None*)
Dumps recording extractor to json file. The extractor can be re-loaded with spikeextractors.load_extractor_from_json(json_file)

file_path: str Path of the json file

relative_to: str, Path, or None If not None, file_paths are serialized relative to this path

dump_to_pickle(*file_path=None*, *include_properties=True*, *include_features=True*, *relative_to=None*)
Dumps recording extractor to a pickle file. The extractor can be re-loaded with spikeextractors.load_extractor_from_json(json_file)

file_path: str Path of the json file

include_properties: bool If True, all properties are dumped

include_features: bool If True, all features are dumped

relative_to: str, Path, or None If not None, file_paths are serialized relative to this path

get_annotation(*annotation_name*)
This function returns the data stored under the annotation name.

annotation_name: str A property stored by the Extractor

annotation_data The data associated with the given property name. Could be many formats as specified by the user

get_annotation_keys()
This function returns a list of stored annotation keys

property_names: list List of stored annotation keys

get_epoch_info(*epoch_name*)
This function returns the start frame and end frame of the epoch in a dict.

epoch_name: str The name of the epoch to be returned

epoch_info: dict A dict containing the start frame and end frame of the epoch

get_epoch_names()
This function returns a list of all the epoch names in the extractor

epoch_names: list List of epoch names in the recording extractor

get_tmp_folder()
Returns temporary folder associated to the extractor

temp_folder: Path The temporary folder

static load_extractor_from_dict(*d*)
Instantiates extractor from dictionary

d: dictionary Python dictionary

extractor: RecordingExtractor or SortingExtractor The loaded extractor object

static load_extractor_from_json(*json_file*)
Instantiates extractor from json file

json_file: str or Path Path to json file

extractor: RecordingExtractor or SortingExtractor The loaded extractor object

static load_extractor_from_pickle(*pkl_file*)
Instantiates extractor from pickle file.

pkl_file: str or Path Path to pickle file

extractor: RecordingExtractor or SortingExtractor The loaded extractor object

make_serialized_dict(*relative_to=None*)
Makes a nested serialized dictionary out of the extractor. The dictionary be used to re-initialize an extractor with spikeextractors.load_extractor_from_dict(dump_dict)

relative_to: str, Path, or None If not None, file_paths are serialized relative to this path

dump_dict: dict Serialized dictionary

remove_epoch(*epoch_name*)
This function removes an epoch from your extractor.

epoch_name: str The name of the epoch to be removed

set_tmp_folder(*folder*)
Sets temporary folder of the extractor

folder: str or Path The temporary folder

class spikeextractors.**RecordingExtractor**
A class that contains functions for extracting important information from recorded extracellular data. It is an abstract class so all functions with the @abstractmethod tag must be implemented for the initialization to work.

clear_channel_gains(*channel_ids=None*)
This function clears the gains of each channel specified by channel_ids

channel_ids: array-like or int The channel ids (ints) for which the groups will be cleared. If None, all channel ids are assumed.

clear_channel_groups(*channel_ids=None*)
This function clears the group of each channel specified by channel_ids

channel_ids: array-like or int The channel ids (ints) for which the groups will be cleared. If None, all channel ids are assumed.

clear_channel_locations(*channel_ids=None*)
This function clears the location of each channel specified by channel_ids.

channel_ids: array-like or int The channel ids (ints) for which the locations will be cleared. If None, all channel ids are assumed.

clear_channel_offsets(*channel_ids=None*)
This function clears the gains of each channel specified by channel_ids.

channel_ids: array-like or int The channel ids (ints) for which the groups will be cleared. If None, all channel ids are assumed.

clear_channel_property(*channel_id*, *property_name*)
This function clears the channel property for the given property.

channel_id: int The id that specifies a channel in the recording

property_name: string The name of the property to be cleared

clear_channels_property(*property_name*, *channel_ids=None*)
This function clears the channels' properties for the given property.

property_name: string The name of the property to be cleared

channel_ids: list A list of ids that specifies a set of channels in the recording. If None all channels are cleared

copy_channel_properties(*recording*, *channel_ids=None*)
Copy channel properties from another recording extractor to the current recording extractor.

recording: RecordingExtractor The recording extractor from which the properties will be copied

channel_ids: (array_like, (int, np.integer)) The list (or single value) of channel_ids for which the properties will be copied

copy_times(*extractor*)
This function copies times from another extractor.

extractor: BaseExtractor The extractor from which the epochs will be copied

frame_to_time(*frames*)
This function converts user-inputted frame indexes to times with units of seconds.

frames: float or array-like The frame or frames to be converted to times

times: float or array-like The corresponding times in seconds

get_channel_gains(*channel_ids=None*)
This function returns the gain of each channel specified by channel_ids.

channel_ids: array_like The channel ids (ints) for which the gains will be returned

gains: array_like Returns a list of corresponding gains (floats) for the given channel_ids

get_channel_groups(*channel_ids=None*)
This function returns the group of each channel specified by channel_ids

channel_ids: array-like or int The channel ids (ints) for which the groups will be returned

groups: array_like Returns a list of corresponding groups (ints) for the given channel_ids

get_channel_ids()
Returns the list of channel ids. If not specified, the range from 0 to num_channels - 1 is returned.

channel_ids: list Channel list

get_channel_locations(*channel_ids=None*, *locations_2d=True*)
This function returns the location of each channel specified by channel_ids

channel_ids: array-like or int The channel ids (ints) for which the locations will be returned. If None, all channel ids are assumed.

locations_2d: bool If True (default), first two dimensions are returned

locations: array_like Returns a list of corresponding locations (floats) for the given channel_ids

get_channel_offsets(*channel_ids=None*)
This function returns the offset of each channel specified by channel_ids.

channel_ids: array_like The channel ids (ints) for which the gains will be returned

offsets: array_like Returns a list of corresponding offsets for the given channel_ids

get_channel_property(*channel_id*, *property_name*)
This function returns the data stored under the property name from the given channel.

channel_id: int The channel id for which the property will be returned

property_name: str A property stored by the RecordingExtractor (location, etc.)

property_data The data associated with the given property name. Could be many formats as specified by the user

get_channel_property_names(*channel_id*)
Get a list of property names for a given channel.

channel_id: int The channel id for which the property names will be returned If None (default), will return property names for all channels

property_names The list of property names

get_dtype(*return_scaled=True*)
This function returns the traces dtype

return_scaled: bool If False and the recording extractor has unscaled traces, it returns the dtype of unscaled traces. If True (default) it returns the dtype of the scaled traces

dtype: np.dtype The dtype of the traces

get_epoch(*epoch_name*)
This function returns a SubRecordingExtractor which is a view to the given epoch

epoch_name: str The name of the epoch to be returned

epoch_extractor: SubRecordingExtractor A SubRecordingExtractor which is a view to the given epoch

get_num_channels()
This function returns the number of channels in the recording.

num_channels: int Number of channels in the recording

get_num_frames()
This function returns the number of frames in the recording

num_frames: int Number of frames in the recording (duration of recording)

get_sampling_frequency()
This function returns the sampling frequency in units of Hz.

fs: float Sampling frequency of the recordings in Hz

get_shared_channel_property_names(*channel_ids=None*)
Get the intersection of channel property names for a given set of channels or for all channels if channel_ids is None.

channel_ids: array_like The channel ids for which the shared property names will be returned. If None (default), will return shared property names for all channels

property_names The list of shared property names

get_snippets(*reference_frames*, *snippet_len*, *channel_ids=None*, *return_scaled=True*)
This function returns data snippets from the given channels that are starting on the given frames and are the length of the given snippet lengths before and after.

reference_frames: array_like A list or array of frames that will be used as the reference frame of each snippet.

snippet_len: int or tuple If int, the snippet will be centered at the reference frame and and return half before and half after of the length. If tuple, it will return the first value of before frames and the second value of after frames around the reference frame (allows for asymmetry).

channel_ids: array_like A list or array of channel ids (ints) from which each trace will be extracted

return_scaled: bool If True, snippets are returned after scaling (using gain/offset). If False, the raw traces are returned.

snippets: numpy.ndarray Returns a list of the snippets as numpy arrays. The length of the list is len(reference_frames) Each array has dimensions: (num_channels x snippet_len) Out-of-bounds cases should be handled by filling in zeros in the snippet

get_sub_extractors_by_property(*property_name*, *return_property_list=False*)
Returns a list of SubRecordingExtractors from this RecordingExtractor based on the given property_name (e.g. group)

property_name: str The property used to subdivide the extractor

return_property_list: bool If True the property list is returned

sub_list: list The list of subextractors to be returned

OR sub_list, prop_list

If return_property_list is True, the property list will be returned as well

get_traces(*channel_ids=None*, *start_frame=None*, *end_frame=None*, *return_scaled=True*)
This function extracts and returns a trace from the recorded data from the given channels ids and the given start and end frame. It will return traces from within three ranges:

[start_frame, start_frame+1, ..., end_frame-1] [start_frame, start_frame+1, ..., final_recording_frame - 1] [0, 1, ..., end_frame-1] [0, 1, ..., final_recording_frame - 1]

if both start_frame and end_frame are given, if only start_frame is given, if only end_frame is given, or if neither start_frame or end_frame are given, respectively. Traces are returned in a 2D array that contains all of the traces from each channel with dimensions (num_channels x num_frames). In this implementation, start_frame is inclusive and end_frame is exclusive conforming to numpy standards.

channel_ids: array_like A list or 1D array of channel ids (ints) from which each trace will be extracted.

start_frame: int The starting frame of the trace to be returned (inclusive).

end_frame: int The ending frame of the trace to be returned (exclusive).

return_scaled: bool If True, traces are returned after scaling (using gain/offset). If False, the raw traces are returned.

traces: numpy.ndarray A 2D array that contains all of the traces from each channel. Dimensions are: (num_channels x num_frames)

get_ttl_events(*start_frame=None*, *end_frame=None*, *channel_id=0*)

Returns an array with frames of TTL signals. To be implemented in sub-classes

start_frame: int The starting frame of the ttl to be returned (inclusive)

end_frame: int The ending frame of the ttl to be returned (exclusive)

channel_id: int The TTL channel id

ttl_frames: array-like Frames of TTL signal for the specified channel

ttl_state: array-like State of the transition: 1 - rising, -1 - falling

load_probe_file(*probe_file*, *channel_map=None*, *channel_groups=None*, *verbose=False*)

This function returns a SubRecordingExtractor that contains information from the given probe file (channel locations, groups, etc.) If a .prb file is given, then 'location' and 'group' information for each channel is added to the SubRecordingExtractor. If a .csv file is given, then it will only add 'location' to the SubRecordingExtractor.

probe_file: str Path to probe file. Either .prb or .csv

channel_map [array-like] A list of channel IDs to set in the loaded file. Only used if the loaded file is a .csv.

channel_groups [array-like] A list of groups (ints) for the channel_ids to set in the loaded file. Only used if the loaded file is a .csv.

verbose: bool If True, output is verbose

subrecording = SubRecordingExtractor The extractor containing all of the probe information.

save_to_probe_file(*probe_file*, *grouping_property=None*, *radius=None*, *graph=True*, *geometry=True*, *verbose=False*)

Saves probe file from the channel information of this recording extractor.

probe_file: str file name of .prb or .csv file to save probe information to

grouping_property: str (default None) If grouping_property is a shared_channel_property, different groups are saved based on the property.

radius: float (default None) Adjacency radius (used by some sorters). If None it is not saved to the probe file.

graph: bool If True, the adjacency graph is saved (default=True)

geometry: bool If True, the geometry is saved (default=True)

verbose: bool If True, output is verbose

set_channel_gains(*gains*, *channel_ids=None*)
This function sets the gain key property of each specified channel id with the corresponding group of the passed in gains float/list.

gains: float/array_like If a float, each channel will be assigned the corresponding gain. If a list, each channel will be given a gain from the list

channel_ids: array_like or None The channel ids (ints) for which the groups will be specified. If None, all channel ids are assumed.

set_channel_groups(*groups*, *channel_ids=None*)
This function sets the group key property of each specified channel id with the corresponding group of the passed in groups list.

groups: array-like or int A list of groups (ints) for the channel_ids

channel_ids: array_like or None The channel ids (ints) for which the groups will be specified. If None, all channel ids are assumed.

set_channel_locations(*locations*, *channel_ids=None*)
This function sets the location key properties of each specified channel id with the corresponding locations of the passed in locations list.

locations: array_like A list of corresponding locations (array_like) for the given channel_ids

channel_ids: array-like or int The channel ids (ints) for which the locations will be specified. If None, all channel ids are assumed.

set_channel_offsets(*offsets*, *channel_ids=None*)
This function sets the offset key property of each specified channel id with the corresponding group of the passed in gains float/list.

offsets: float/array_like If a float, each channel will be assigned the corresponding offset. If a list, each channel will be given an offset from the list

channel_ids: array_like or None The channel ids (ints) for which the groups will be specified. If None, all channel ids are assumed.

set_channel_property(*channel_id*, *property_name*, *value*)
This function adds a property dataset to the given channel under the property name.

channel_id: int The channel id for which the property will be added

property_name: str A property stored by the RecordingExtractor (location, etc.)

value: The data associated with the given property name. Could be many formats as specified by the user

set_times(*times*)
This function sets the recording times (in seconds) for each frame

times: array-like The times in seconds for each frame

time_to_frame(*times*)
This function converts a user-inputted times (in seconds) to a frame indexes.

times: float or array-like The times (in seconds) to be converted to frame indexes

frames: float or array-like The corresponding frame indexes

static write_recording(*recording*, *save_path*)

This function writes out the recorded file of a given recording extractor to the file format of this current recording extractor. Allows for easy conversion between recording file formats. It is a static method so it can be used without instantiating this recording extractor.

recording: RecordingExtractor An RecordingExtractor that can extract information from the recording file to be converted to the new format.

save_path: string A path to where the converted recorded data will be saved, which may either be a file or a folder, depending on the format.

write_to_binary_dat_format(*save_path*, *time_axis=0*, *dtype=None*, *chunk_size=None*, *chunk_mb=500*, *n_jobs=1*, *joblib_backend='loky'*, *return_scaled=True*, *verbose=False*)

Saves the traces of this recording extractor into binary .dat format.

save_path: str The path to the file.

time_axis: 0 (default) or 1 If 0 then traces are transposed to ensure (nb_sample, nb_channel) in the file. If 1, the traces shape (nb_channel, nb_sample) is kept in the file.

dtype: dtype Type of the saved data. Default float32

chunk_size: None or int Size of each chunk in number of frames. If None (default) and 'chunk_mb' is given, the file is saved in chunks of 'chunk_mb' Mb (default 500Mb)

chunk_mb: None or int Chunk size in Mb (default 500Mb)

n_jobs: int Number of jobs to use (Default 1)

joblib_backend: str Joblib backend for parallel processing ('loky', 'threading', 'multiprocessing')

return_scaled: bool If True, traces are returned after scaling (using gain/offset). If False, the raw traces are returned

verbose: bool If True, output is verbose (when chunks are used)

write_to_h5_dataset_format(*dataset_path*, *save_path=None*, *file_handle=None*, *time_axis=0*, *dtype=None*, *chunk_size=None*, *chunk_mb=500*, *verbose=False*)

Saves the traces of a recording extractor in an h5 dataset.

dataset_path: str Path to dataset in h5 file (e.g. '/dataset')

save_path: str The path to the file.

file_handle: file handle The file handle to dump data. This can be used to append data to an header. In case file_handle is given, the file is NOT closed after writing the binary data.

time_axis: 0 (default) or 1 If 0 then traces are transposed to ensure (nb_sample, nb_channel) in the file. If 1, the traces shape (nb_channel, nb_sample) is kept in the file.

dtype: dtype Type of the saved data. Default float32.

chunk_size: None or int Size of each chunk in number of frames. If None (default) and 'chunk_mb' is given, the file is saved in chunks of 'chunk_mb' Mb (default 500Mb)

chunk_mb: None or int Chunk size in Mb (default 500Mb)

verbose: bool If True, output is verbose (when chunks are used)

class spikeextractors.**SortingExtractor**

A class that contains functions for extracting important information from spiked sorted data given a spike sorting software. It is an abstract class so all functions with the @abstractmethod tag must be implemented for the initialization to work.

clear_unit_property(*unit_id*, *property_name*)
This function clears the unit property for the given property.

unit_id: int The id that specifies a unit in the sorting

property_name: string The name of the property to be cleared

clear_unit_spike_features(*unit_id*, *feature_name*)
This function clears the unit spikes features for the given feature.

unit_id: int The id that specifies a unit in the sorting

feature_name: string The name of the feature to be cleared

clear_units_property(*property_name*, *unit_ids=None*)
This function clears the units' properties for the given property.

property_name: string The name of the property to be cleared

unit_ids: list A list of ids that specifies a set of units in the sorting. If None, all units are cleared

clear_units_spike_features(*feature_name*, *unit_ids=None*)
This function clears the units' spikes features for the given feature.

feature_name: string The name of the feature to be cleared

unit_ids: list A list of ids that specifies a set of units in the sorting. If None, all units are cleared

copy_times(*extractor*)
This function copies times from another extractor.

extractor: BaseExtractor The extractor from which the epochs will be copied

copy_unit_properties(*sorting*, *unit_ids=None*)
Copy unit properties from another sorting extractor to the current sorting extractor.

sorting: SortingExtractor The sorting extractor from which the properties will be copied

unit_ids: (array_like, (int, np.integer)) The list (or single value) of unit_ids for which the properties will be copied

copy_unit_spike_features(*sorting*, *unit_ids=None*)
Copy unit spike features from another sorting extractor to the current sorting extractor.

sorting: SortingExtractor The sorting extractor from which the spike features will be copied

unit_ids: (array_like, (int, np.integer)) The list (or single value) of unit_ids for which the spike features will be copied

frame_to_time(*frames*)
This function converts user-inputted frame indexes to times with units of seconds.

frames: float or array-like The frame or frames to be converted to times

times: float or array-like The corresponding times in seconds

get_epoch(*epoch_name*)
This function returns a SubSortingExtractor which is a view to the given epoch.

epoch_name: str The name of the epoch to be returned

epoch_extractor: SubRecordingExtractor A SubRecordingExtractor which is a view to the given epoch

get_sampling_frequency()
It returns the sampling frequency.

sampling_frequency: float The sampling frequency

get_shared_unit_property_names(*unit_ids=None*)
Get the intersection of unit property names for a given set of units or for all units if unit_ids is None.

unit_ids: array_like The unit ids for which the shared property names will be returned. If None (default), will return shared property names for all units

property_names The list of shared property names

get_shared_unit_spike_feature_names(*unit_ids=None*)
Get the intersection of unit feature names for a given set of units or for all units if unit_ids is None.

unit_ids: array_like The unit ids for which the shared feature names will be returned. If None (default), will return shared feature names for all units

property_names The list of shared feature names

get_sub_extractors_by_property(*property_name*, *return_property_list=False*)
Returns a list of SubSortingExtractors from this SortingExtractor based on the given property_name (e.g. group)

property_name: str The property used to subdivide the extractor

return_property_list: bool If True the property list is returned

sub_list: list The list of subextractors to be returned

get_unit_ids()
This function returns a list of ids (ints) for each unit in the sorsted result.

unit_ids: array_like A list of the unit ids in the sorted result (ints).

get_unit_property(*unit_id*, *property_name*)
This function returns the data stored under the property name given from the given unit.

unit_id: int The unit id for which the property will be returned

property_name: str The name of the property

value The data associated with the given property name. Could be many formats as specified by the user

get_unit_property_names(*unit_id*)
Get a list of property names for a given unit.

unit_id: int The unit id for which the property names will be returned

property_names The list of property names

get_unit_spike_feature_names(*unit_id*)
This function returns the list of feature names for the given unit

unit_id: int The unit id for which the feature names will be returned

property_names The list of feature names.

get_unit_spike_features(*unit_id*, *feature_name*, *start_frame=None*, *end_frame=None*)
This function extracts the specified spike features from the specified unit. It will return spike features from within three ranges:

> [start_frame, t_start+1, ..., end_frame-1] [start_frame, start_frame+1, ..., final_unit_spike_frame - 1] [0, 1, ..., end_frame-1] [0, 1, ..., final_unit_spike_frame - 1]

if both start_frame and end_frame are given, if only start_frame is given, if only end_frame is given, or if neither start_frame or end_frame are given, respectively. Spike features are returned in the form of an array_like of spike features. In this implementation, start_frame is inclusive and end_frame is exclusive conforming to numpy standards.

unit_id: int The id that specifies a unit in the recording

feature_name: string The name of the feature to be returned

start_frame: int The frame above which a spike frame is returned (inclusive)

end_frame: int The frame below which a spike frame is returned (exclusive)

spike_features: numpy.ndarray An array containing all the features for each spike in the specified unit given the range of start and end frames

get_unit_spike_train(*unit_id*, *start_frame=None*, *end_frame=None*)
This function extracts spike frames from the specified unit. It will return spike frames from within three ranges:

> [start_frame, t_start+1, ..., end_frame-1] [start_frame, start_frame+1, ..., final_unit_spike_frame - 1] [0, 1, ..., end_frame-1] [0, 1, ..., final_unit_spike_frame - 1]

if both start_frame and end_frame are given, if only start_frame is given, if only end_frame is given, or if neither start_frame or end_frame are given, respectively. Spike frames are returned in the form of an array_like of spike frames. In this implementation, start_frame is inclusive and end_frame is exclusive conforming to numpy standards.

unit_id: int The id that specifies a unit in the recording

start_frame: int The frame above which a spike frame is returned (inclusive)

end_frame: int The frame below which a spike frame is returned (exclusive)

spike_train: numpy.ndarray An 1D array containing all the frames for each spike in the specified unit given the range of start and end frames

get_units_property(*, *unit_ids=None*, *property_name*)
Returns a list of values stored under the property name corresponding to a list of units

unit_ids: list The unit ids for which the property will be returned Defaults to get_unit_ids()

property_name: str The name of the property

values The list of values

get_units_spike_train(*unit_ids=None*, *start_frame=None*, *end_frame=None*)
This function extracts spike frames from the specified units.

unit_ids: array_like The unit ids from which to return spike trains. If None, all unit spike trains will be returned

start_frame: int The frame above which a spike frame is returned (inclusive)

end_frame: int The frame below which a spike frame is returned (exclusive)

spike_train: numpy.ndarray An 2D array containing all the frames for each spike in the specified units given the range of start and end frames

get_unsorted_spike_train(*start_frame=None*, *end_frame=None*)
This function extracts spike frames from the unsorted events. It will return spike frames from within three ranges:

[start_frame, t_start+1, ..., end_frame-1] [start_frame, start_frame+1, ..., final_unit_spike_frame - 1] [0, 1, ..., end_frame-1] [0, 1, ..., final_unit_spike_frame - 1]

if both start_frame and end_frame are given, if only start_frame is given, if only end_frame is given, or if neither start_frame or end_frame are given, respectively. Spike frames are returned in the form of an array_like of spike frames. In this implementation, start_frame is inclusive and end_frame is exclusive conforming to numpy standards.

start_frame: int The frame above which a spike frame is returned (inclusive)

end_frame: int The frame below which a spike frame is returned (exclusive)

spike_train: numpy.ndarray An 1D array containing all the frames for each spike in the specified unit given the range of start and end frames

set_sampling_frequency(*sampling_frequency*)
It sets the sorting extractor sampling frequency.

sampling_frequency: float The sampling frequency

set_times(*times*)
This function sets the sorting times to convert spike trains to seconds

times: array-like The times in seconds for each frame

set_unit_property(*unit_id*, *property_name*, *value*)
This function adds a unit property data set under the given property name to the given unit.

unit_id: int The unit id for which the property will be set

property_name: str The name of the property to be stored

value The data associated with the given property name. Could be many formats as specified by the user

set_unit_spike_features(*unit_id*, *feature_name*, *value*, *indexes=None*)
This function adds a unit features data set under the given features name to the given unit.

unit_id: int The unit id for which the features will be set

feature_name: str The name of the feature to be stored

value: array_like The data associated with the given feature name. Could be many formats as specified by the user.

indexes: array_like The indices of the specified spikes (if the number of spike features is less than the length of the unit's spike train). If None, it is assumed that value has the same length as the spike train.

set_units_property(*, *unit_ids=None*, *property_name*, *values*)
Sets unit property data for a list of units

unit_ids: list The list of unit ids for which the property will be set Defaults to get_unit_ids()

property_name: str The name of the property

value: list The list of values to be set

time_to_frame(*times*)
This function converts a user-inputted times (in seconds) to a frame indexes.

times: float or array-like The times (in seconds) to be converted to frame indexes

frames: float or array-like The corresponding frame indexes

static write_sorting(*sorting*, *save_path*)
This function writes out the spike sorted data file of a given sorting extractor to the file format of this current sorting extractor. Allows for easy conversion between spike sorting file formats. It is a static method so it can be used without instantiating this sorting extractor.

sorting: SortingExtractor A SortingExtractor that can extract information from the sorted data file to be converted to the new format

save_path: string A path to where the converted sorted data will be saved, which may either be a file or a folder, depending on the format

class spikeextractors.**SubRecordingExtractor**(*parent_recording*, *, *channel_ids=None*, *renamed_channel_ids=None*, *start_frame=None*, *end_frame=None*)

copy_channel_properties(*recording*, *channel_ids=None*)
Copy channel properties from another recording extractor to the current recording extractor.

recording: RecordingExtractor The recording extractor from which the properties will be copied

channel_ids: (array_like, (int, np.integer)) The list (or single value) of channel_ids for which the properties will be copied

frame_to_time(*frame*)
This function converts user-inputted frame indexes to times with units of seconds.

frames: float or array-like The frame or frames to be converted to times

times: float or array-like The corresponding times in seconds

get_channel_ids()
Returns the list of channel ids. If not specified, the range from 0 to num_channels - 1 is returned.

channel_ids: list Channel list

get_num_frames()
This function returns the number of frames in the recording

num_frames: int Number of frames in the recording (duration of recording)

get_sampling_frequency()
This function returns the sampling frequency in units of Hz.

fs: float Sampling frequency of the recordings in Hz

get_snippets(*reference_frames*, *snippet_len*, *channel_ids=None*, *return_scaled=True*)
This function returns data snippets from the given channels that are starting on the given frames and are the length of the given snippet lengths before and after.

reference_frames: array_like A list or array of frames that will be used as the reference frame of each snippet.

snippet_len: int or tuple If int, the snippet will be centered at the reference frame and and return half before and half after of the length. If tuple, it will return the first value of before frames and the second value of after frames around the reference frame (allows for asymmetry).

channel_ids: array_like A list or array of channel ids (ints) from which each trace will be extracted

return_scaled: bool If True, snippets are returned after scaling (using gain/offset). If False, the raw traces are returned.

snippets: numpy.ndarray Returns a list of the snippets as numpy arrays. The length of the list is len(reference_frames) Each array has dimensions: (num_channels x snippet_len) Out-of-bounds cases should be handled by filling in zeros in the snippet

get_traces(*channel_ids=None*, *start_frame=None*, *end_frame=None*, *return_scaled=True*)

This function extracts and returns a trace from the recorded data from the given channels ids and the given start and end frame. It will return traces from within three ranges:

[start_frame, start_frame+1, ..., end_frame-1] [start_frame, start_frame+1, ..., final_recording_frame - 1] [0, 1, ..., end_frame-1] [0, 1, ..., final_recording_frame - 1]

if both start_frame and end_frame are given, if only start_frame is given, if only end_frame is given, or if neither start_frame or end_frame are given, respectively. Traces are returned in a 2D array that contains all of the traces from each channel with dimensions (num_channels x num_frames). In this implementation, start_frame is inclusive and end_frame is exclusive conforming to numpy standards.

channel_ids: array_like A list or 1D array of channel ids (ints) from which each trace will be extracted.

start_frame: int The starting frame of the trace to be returned (inclusive).

end_frame: int The ending frame of the trace to be returned (exclusive).

return_scaled: bool If True, traces are returned after scaling (using gain/offset). If False, the raw traces are returned.

traces: numpy.ndarray A 2D array that contains all of the traces from each channel. Dimensions are: (num_channels x num_frames)

get_ttl_events(*start_frame=None*, *end_frame=None*, *channel_id=0*)

Returns an array with frames of TTL signals. To be implemented in sub-classes

start_frame: int The starting frame of the ttl to be returned (inclusive)

end_frame: int The ending frame of the ttl to be returned (exclusive)

channel_id: int The TTL channel id

ttl_frames: array-like Frames of TTL signal for the specified channel

ttl_state: array-like State of the transition: 1 - rising, -1 - falling

time_to_frame(*time*)

This function converts a user-inputted times (in seconds) to a frame indexes.

times: float or array-like The times (in seconds) to be converted to frame indexes

frames: float or array-like The corresponding frame indexes

class spikeextractors.**SubSortingExtractor**(*parent_sorting*, *, *unit_ids=None*, *renamed_unit_ids=None*, *start_frame=None*, *end_frame=None*)

copy_unit_properties(*sorting*, *unit_ids=None*)
Copy unit properties from another sorting extractor to the current sorting extractor.

sorting: SortingExtractor The sorting extractor from which the properties will be copied

unit_ids: (array_like, (int, np.integer)) The list (or single value) of unit_ids for which the properties will be copied

copy_unit_spike_features(*sorting*, *unit_ids=None*, *start_frame=None*, *end_frame=None*)
Copy unit spike features from another sorting extractor to the current sorting extractor.

sorting: SortingExtractor The sorting extractor from which the spike features will be copied

unit_ids: (array_like, (int, np.integer)) The list (or single value) of unit_ids for which the spike features will be copied

frame_to_time(*frame*)
This function converts user-inputted frame indexes to times with units of seconds.

frames: float or array-like The frame or frames to be converted to times

times: float or array-like The corresponding times in seconds

get_sampling_frequency()
It returns the sampling frequency.

sampling_frequency: float The sampling frequency

get_unit_ids()
This function returns a list of ids (ints) for each unit in the sorsted result.

unit_ids: array_like A list of the unit ids in the sorted result (ints).

get_unit_spike_train(*unit_id*, *start_frame=None*, *end_frame=None*)
This function extracts spike frames from the specified unit. It will return spike frames from within three ranges:

[start_frame, t_start+1, ..., end_frame-1] [start_frame, start_frame+1, ..., final_unit_spike_frame - 1] [0, 1, ..., end_frame-1] [0, 1, ..., final_unit_spike_frame - 1]

if both start_frame and end_frame are given, if only start_frame is given, if only end_frame is given, or if neither start_frame or end_frame are given, respectively. Spike frames are returned in the form of an array_like of spike frames. In this implementation, start_frame is inclusive and end_frame is exclusive conforming to numpy standards.

unit_id: int The id that specifies a unit in the recording

start_frame: int The frame above which a spike frame is returned (inclusive)

end_frame: int The frame below which a spike frame is returned (exclusive)

spike_train: numpy.ndarray An 1D array containing all the frames for each spike in the specified unit given the range of start and end frames

time_to_frame(*time*)
This function converts a user-inputted times (in seconds) to a frame indexes.

times: float or array-like The times (in seconds) to be converted to frame indexes

frames: float or array-like The corresponding frame indexes

class spikeextractors.**MultiRecordingChannelExtractor**(*recordings*, *groups=None*)

get_channel_ids()
Returns the list of channel ids. If not specified, the range from 0 to num_channels - 1 is returned.

channel_ids: list Channel list

get_num_frames()
This function returns the number of frames in the recording

num_frames: int Number of frames in the recording (duration of recording)

get_sampling_frequency()
This function returns the sampling frequency in units of Hz.

fs: float Sampling frequency of the recordings in Hz

get_traces(*channel_ids=None*, *start_frame=None*, *end_frame=None*, *return_scaled=True*)
This function extracts and returns a trace from the recorded data from the given channels ids and the given start and end frame. It will return traces from within three ranges:

[start_frame, start_frame+1, ..., end_frame-1] [start_frame, start_frame+1, ..., final_recording_frame - 1] [0, 1, ..., end_frame-1] [0, 1, ..., final_recording_frame - 1]

if both start_frame and end_frame are given, if only start_frame is given, if only end_frame is given, or if neither start_frame or end_frame are given, respectively. Traces are returned in a 2D array that contains all of the traces from each channel with dimensions (num_channels x num_frames). In this implementation, start_frame is inclusive and end_frame is exclusive conforming to numpy standards.

channel_ids: array_like A list or 1D array of channel ids (ints) from which each trace will be extracted.

start_frame: int The starting frame of the trace to be returned (inclusive).

end_frame: int The ending frame of the trace to be returned (exclusive).

return_scaled: bool If True, traces are returned after scaling (using gain/offset). If False, the raw traces are returned.

traces: numpy.ndarray A 2D array that contains all of the traces from each channel. Dimensions are: (num_channels x num_frames)

class spikeextractors.**MultiRecordingTimeExtractor**(*recordings*, *epoch_names=None*)

frame_to_time(*frame*)
This function converts user-inputted frame indexes to times with units of seconds.

frames: float or array-like The frame or frames to be converted to times

times: float or array-like The corresponding times in seconds

get_channel_ids()
Returns the list of channel ids. If not specified, the range from 0 to num_channels - 1 is returned.

channel_ids: list Channel list

get_num_frames()
This function returns the number of frames in the recording

num_frames: int Number of frames in the recording (duration of recording)

get_sampling_frequency()
This function returns the sampling frequency in units of Hz.

fs: float Sampling frequency of the recordings in Hz

get_traces(*channel_ids=None*, *start_frame=None*, *end_frame=None*, *return_scaled=True*)
This function extracts and returns a trace from the recorded data from the given channels ids and the given start and end frame. It will return traces from within three ranges:

[start_frame, start_frame+1, ..., end_frame-1] [start_frame, start_frame+1, ..., final_recording_frame - 1] [0, 1, ..., end_frame-1] [0, 1, ..., final_recording_frame - 1]

if both start_frame and end_frame are given, if only start_frame is given, if only end_frame is given, or if neither start_frame or end_frame are given, respectively. Traces are returned in a 2D array that contains all of the traces from each channel with dimensions (num_channels x num_frames). In this implementation, start_frame is inclusive and end_frame is exclusive conforming to numpy standards.

channel_ids: array_like A list or 1D array of channel ids (ints) from which each trace will be extracted.

start_frame: int The starting frame of the trace to be returned (inclusive).

end_frame: int The ending frame of the trace to be returned (exclusive).

return_scaled: bool If True, traces are returned after scaling (using gain/offset). If False, the raw traces are returned.

traces: numpy.ndarray A 2D array that contains all of the traces from each channel. Dimensions are: (num_channels x num_frames)

get_ttl_events(*start_frame=None*, *end_frame=None*, *channel_id=0*)
Returns an array with frames of TTL signals. To be implemented in sub-classes

start_frame: int The starting frame of the ttl to be returned (inclusive)

end_frame: int The ending frame of the ttl to be returned (exclusive)

channel_id: int The TTL channel id

ttl_frames: array-like Frames of TTL signal for the specified channel

ttl_state: array-like State of the transition: 1 - rising, -1 - falling

time_to_frame(*time*)
This function converts a user-inputted times (in seconds) to a frame indexes.

times: float or array-like The times (in seconds) to be converted to frame indexes

frames: float or array-like The corresponding frame indexes

class spikeextractors.**MultiSortingExtractor**(*sortings*)

clear_unit_property(*unit_id*, *property_name*)
This function clears the unit property for the given property.

unit_id: int The id that specifies a unit in the sorting

property_name: string The name of the property to be cleared

clear_unit_spike_features(*unit_id*, *feature_name*)
This function clears the unit spikes features for the given feature.

unit_id: int The id that specifies a unit in the sorting

feature_name: string The name of the feature to be cleared

get_sampling_frequency()
It returns the sampling frequency.

sampling_frequency: float The sampling frequency

get_unit_ids()
This function returns a list of ids (ints) for each unit in the sorsted result.

unit_ids: array_like A list of the unit ids in the sorted result (ints).

get_unit_property(*unit_id*, *property_name*)
This function returns the data stored under the property name given from the given unit.

unit_id: int The unit id for which the property will be returned

property_name: str The name of the property

value The data associated with the given property name. Could be many formats as specified by the user

get_unit_property_names(*unit_id*)
Get a list of property names for a given unit.

unit_id: int The unit id for which the property names will be returned

property_names The list of property names

get_unit_spike_feature_names(*unit_id*)
This function returns the list of feature names for the given unit

unit_id: int The unit id for which the feature names will be returned

property_names The list of feature names.

get_unit_spike_features(*unit_id*, *feature_name*, *start_frame=None*, *end_frame=None*)
This function extracts the specified spike features from the specified unit. It will return spike features from within three ranges:

> [start_frame, t_start+1, ..., end_frame-1] [start_frame, start_frame+1, ..., final_unit_spike_frame - 1] [0, 1, ..., end_frame-1] [0, 1, ..., final_unit_spike_frame - 1]

if both start_frame and end_frame are given, if only start_frame is given, if only end_frame is given, or if neither start_frame or end_frame are given, respectively. Spike features are returned in the form of an array_like of spike features. In this implementation, start_frame is inclusive and end_frame is exclusive conforming to numpy standards.

unit_id: int The id that specifies a unit in the recording

feature_name: string The name of the feature to be returned

start_frame: int The frame above which a spike frame is returned (inclusive)

end_frame: int The frame below which a spike frame is returned (exclusive)

spike_features: numpy.ndarray An array containing all the features for each spike in the specified unit given the range of start and end frames

get_unit_spike_train(*unit_id*, *start_frame=None*, *end_frame=None*)
This function extracts spike frames from the specified unit. It will return spike frames from within three ranges:

[start_frame, t_start+1, ..., end_frame-1] [start_frame, start_frame+1, ..., final_unit_spike_frame - 1] [0, 1, ..., end_frame-1] [0, 1, ..., final_unit_spike_frame - 1]

if both start_frame and end_frame are given, if only start_frame is given, if only end_frame is given, or if neither start_frame or end_frame are given, respectively. Spike frames are returned in the form of an array_like of spike frames. In this implementation, start_frame is inclusive and end_frame is exclusive conforming to numpy standards.

unit_id: int The id that specifies a unit in the recording

start_frame: int The frame above which a spike frame is returned (inclusive)

end_frame: int The frame below which a spike frame is returned (exclusive)

spike_train: numpy.ndarray An 1D array containing all the frames for each spike in the specified unit given the range of start and end frames

set_sampling_frequency(*sampling_frequency*)
It sets the sorting extractor sampling frequency.

sampling_frequency: float The sampling frequency

set_unit_property(*unit_id*, *property_name*, *value*)
This function adds a unit property data set under the given property name to the given unit.

unit_id: int The unit id for which the property will be set

property_name: str The name of the property to be stored

value The data associated with the given property name. Could be many formats as specified by the user

set_unit_spike_features(*unit_id*, *feature_name*, *value*, *indexes=None*)
This function adds a unit features data set under the given features name to the given unit.

unit_id: int The unit id for which the features will be set

feature_name: str The name of the feature to be stored

value: array_like The data associated with the given feature name. Could be many formats as specified by the user.

indexes: array_like The indices of the specified spikes (if the number of spike features is less than the length of the unit's spike train). If None, it is assumed that value has the same length as the spike train.

spikeextractors.**load_extractor_from_dict**(*d*)
Instantiates extractor from dictionary

d: dictionary Python dictionary

extractor: RecordingExtractor or SortingExtractor The loaded extractor object

spikeextractors.**load_extractor_from_json**(*json_file*)
Instantiates extractor from json file

json_file: str or Path Path to json file

extractor: RecordingExtractor or SortingExtractor The loaded extractor object

spikeextractors.**load_extractor_from_pickle**(*pkl_file*)
Instantiates extractor from pickle file

pkl_file: str or Path Path to pickle file

extractor: RecordingExtractor or SortingExtractor The loaded extractor object

spikeextractors.**load_probe_file**(*recording*, *probe_file*, *channel_map=None*, *channel_groups=None*, *verbose=False*)
This function returns a SubRecordingExtractor that contains information from the given probe file (channel locations, groups, etc.) If a .prb file is given, then 'location' and 'group' information for each channel is added to the SubRecordingExtractor. If a .csv file is given, then it will only add 'location' to the SubRecordingExtractor.

recording: RecordingExtractor The recording extractor to load channel information from.

probe_file: str Path to probe file. Either .prb or .csv

channel_map [array-like] A list of channel IDs to set in the loaded file. Only used if the loaded file is a .csv.

channel_groups [array-like] A list of groups (ints) for the channel_ids to set in the loaded file. Only used if the loaded file is a .csv.

verbose: bool If True, output is verbose

subrecording: SubRecordingExtractor The extractor containing all of the probe information.

spikeextractors.**save_to_probe_file**(*recording*, *probe_file*, *grouping_property=None*, *radius=None*, *graph=True*, *geometry=True*, *verbose=False*)
Saves probe file from the channel information of the given recording extractor.

recording: RecordingExtractor The recording extractor to save probe file from

probe_file: str file name of .prb or .csv file to save probe information to

grouping_property: str (default None) If grouping_property is a shared_channel_property, different groups are saved based on the property.

radius: float (default None) Adjacency radius (used by some sorters). If None it is not saved to the probe file.

graph: bool If True, the adjacency graph is saved (default=True)

geometry: bool If True, the geometry is saved (default=True)

verbose: bool If True, output is verbose

spikeextractors.**write_to_binary_dat_format**(*recording*, *save_path=None*, *file_handle=None*, *time_axis=0*, *dtype=None*, *chunk_size=None*, *chunk_mb=500*, *n_jobs=1*, *joblib_backend='loky'*, *return_scaled=True*, *verbose=False*)
Saves the traces of a recording extractor in binary .dat format.

recording: RecordingExtractor The recording extractor object to be saved in .dat format

save_path: str The path to the file.

file_handle: file handle The file handle to dump data. This can be used to append data to an header. In case file_handle is given, the file is NOT closed after writing the binary data.

time_axis: 0 (default) or 1 If 0 then traces are transposed to ensure (nb_sample, nb_channel) in the file. If 1, the traces shape (nb_channel, nb_sample) is kept in the file.

dtype: dtype Type of the saved data. Default float32.

chunk_size: None or int Size of each chunk in number of frames. If None (default) and 'chunk_mb' is given, the file is saved in chunks of 'chunk_mb' Mb (default 500Mb)

chunk_mb: None or int Chunk size in Mb (default 500Mb)

n_jobs: int Number of jobs to use (Default 1)

joblib_backend: str Joblib backend for parallel processing ('loky', 'threading', 'multiprocessing')

return_scaled: bool If True, traces are written after scaling (using gain/offset). If False, the raw traces are written

verbose: bool If True, output is verbose (when chunks are used)

spikeextractors.**get_sub_extractors_by_property**(*extractor*, *property_name*, *return_property_list=False*)

Returns a list of SubExtractors from the Extractor based on the given property_name (e.g. group)

extractor: RecordingExtractor or SortingExtractor The extractor object to access SubRecordingExtractors from.

property_name: str The property used to subdivide the extractor

return_property_list: bool If True the property list is returned

sub_list: list The list of subextractors to be returned.

OR sub_list, prop_list

If return_property_list is True, the property list will be returned as well.

9.2 Module spikeinterface.toolkit

9.2.1 Preprocessing

spiketoolkit.preprocessing.**bandpass_filter**(*recording*, *freq_min=300*, *freq_max=6000*, *freq_wid=1000*, *filter_type='fft'*, *order=3*, *chunk_size=30000*, *cache_chunks=False*, *dtype=None*)

Performs a lazy filter on the recording extractor traces.

recording: RecordingExtractor The recording extractor to be filtered.

freq_min: int or float High-pass cutoff frequency.

freq_max: int or float Low-pass cutoff frequency.

freq_wid: int or float Width of the filter (when type is 'fft').

filter_type: str 'fft' or 'butter'. The 'fft' filter uses a kernel in the frequency domain. The 'butter' filter uses scipy butter and filtfilt functions.

order: int Order of the filter (if 'butter').

chunk_size: int The chunk size to be used for the filtering.

cache_chunks: bool (default False). If True then each chunk is cached in memory (in a dict)

dtype: dtype The dtype of the traces

filter_recording: BandpassFilterRecording The filtered recording extractor object

spiketoolkit.preprocessing.**blank_saturation**(*recording*, *threshold=None*, *seed=0*)

Find and remove parts of the signal with extereme values. Some arrays may produce these when amplifiers enter saturation, typically for short periods of time. To remove these artefacts, values below or above a threshold are set to the median signal value. The threshold is either be estimated automatically, using the lower and upper 0.1 signal percentile with the largest deviation from the median, or specificed. Use this function with caution, as it may clip uncontaminated signals. A warning is printed if the data range suggests no artefacts.

recording: RecordingExtractor The recording extractor to be transformed Minimum value. If *None*, clipping is not performed on lower interval edge.

threshold: float or 'None' (default *None*) Threshold value (in absolute units) for saturation artifacts. If *None*, the threshold will be determined from the 0.1 signal percentile.

seed: int Random seed for reproducibility

rescaled_traces: BlankSaturationRecording The filtered traces recording extractor object

spiketoolkit.preprocessing.**clip**(*recording*, *a_min=None*, *a_max=None*)

Limit the values of the data between a_min and a_max. Values exceeding the range will be set to the minimum or maximum, respectively.

recording: RecordingExtractor The recording extractor to be transformed

a_min: float or *None* (default *None*) Minimum value. If *None*, clipping is not performed on lower interval edge.

a_max: float or *None* (default *None*) Maximum value. If *None*, clipping is not performed on upper interval edge.

rescaled_traces: ClipTracesRecording The clipped traces recording extractor object

spiketoolkit.preprocessing.**normalize_by_quantile**(*recording*, *scale=1.0*, *median=0.0*, *q1=0.01*, *q2=0.99*, *seed=0*)

Rescale the traces from the given recording extractor with a scalar and offset. First, the median and quantiles of the distribution are estimated. Then the distribution is rescaled and offset so that the scale is given by the distance between the quantiles (1st and 99th by default) is set to *scale*, and the median is set to the given median.

recording: RecordingExtractor The recording extractor to be transformed

scalar: float Scale for the output distribution

median: float Median for the output distribution

q1: float (default 0.01) Lower quantile used for measuring the scale

q1: float (default 0.99) Upper quantile used for measuring the

seed: int Random seed for reproducibility

rescaled_traces: NormalizeByQuantileRecording The rescaled traces recording extractor object

spiketoolkit.preprocessing.**notch_filter**(*recording*, *freq=3000*, *q=30*, *chunk_size=30000*, *cache_chunks=False*)

Performs a notch filter on the recording extractor traces using scipy iirnotch function.

recording: RecordingExtractor The recording extractor to be notch-filtered.

freq: int or float The target frequency of the notch filter.

q: int The quality factor of the notch filter.

chunk_size: int The chunk size to be used for the filtering.

cache_chunks: bool (default False). If True then each chunk is cached in memory (in a dict)

filter_recording: NotchFilterRecording The notch-filtered recording extractor object

spiketoolkit.preprocessing.**rectify**(*recording*)

Rectifies the recording extractor traces. It is useful, in combination with 'resample', to compute multi-unit activity (MUA).

recording: RecordingExtractor The recording extractor object to be rectified

rectified_recording: RectifyRecording The rectified recording extractor object

spiketoolkit.preprocessing.**remove_artifacts**(*recording*, *triggers*, *ms_before=0.5*, *ms_after=3*, *mode='zeros'*, *fit_sample_spacing=1.0*)

Removes stimulation artifacts from recording extractor traces. By default, artifact periods are zeroed-out (mode = 'zeros'). This is only recommended for traces that are centered around zero (e.g. through a prior highpass filter); if this is not the case, linear and cubic interpolation modes are also available, controlled by the 'mode' input argument.

recording: RecordingExtractor The recording extractor to remove artifacts from

triggers: list List of int with the stimulation trigger frames

ms_before: float Time interval in ms to remove before the trigger events

ms_after: float Time interval in ms to remove after the trigger events

mode: str Determines what artifacts are replaced by. Can be one of the following:

- 'zeros' (default): Artifacts are replaced by zeros.
- **'linear': Replacement are obtained through Linear interpolation between** the trace before and after the artifact. If the trace starts or ends with an artifact period, the gap is filled with the closest available value before or after the artifact.
- **'cubic': Cubic spline interpolation between the trace before and after** the artifact, referenced to evenly spaced fit points before and after the artifact. This is an option thatcan be helpful if there are significant LFP effects around the time of the artifact, but visual inspection of fit behaviour with your chosen settings is recommended. The spacing of fit points is controlled by 'fit_sample_spacing', with greater spacing between points leading to a fit that is less sensitive to high frequency fluctuations but at the cost of a less smooth continuation of the trace. If the trace starts or ends with an artifact, the gap is filled with the closest available value before or after the artifact.

fit_sample_spacing: float Determines the spacing (in ms) of reference points for the cubic spline fit if mode = 'cubic'. Default = 1ms. Note: The actual fit samples are the median of the 5 data points around the time of each sample point to avoid excessive influence from hyper-local fluctuations.

removed_recording: RemoveArtifactsRecording The recording extractor after artifact removal

spiketoolkit.preprocessing.**remove_bad_channels**(*recording*, *bad_channel_ids=None*, *bad_threshold=2*, *seconds=10*, *verbose=False*)

Remove bad channels from the recording extractor.

recording: RecordingExtractor The recording extractor object

bad_channel_ids: list List of bad channel ids (int). If None, automatic removal will be done based on standard deviation.

bad_threshold: float If automatic is used, the threshold for the standard deviation over which channels are removed

seconds: float If automatic is used, the number of seconds used to compute standard deviations

verbose: bool If True, output is verbose

remove_bad_channels_recording: RemoveBadChannelsRecording The recording extractor without bad channels

spiketoolkit.preprocessing.**resample**(*recording*, *resample_rate*)

Resamples the recording extractor traces. If the resampling rate is multiple of the sampling rate, the faster scipy decimate function is used.

recording: RecordingExtractor The recording extractor to be resampled

resample_rate: int or float The resampling frequency

resampled_recording: ResampleRecording The resample recording extractor

spiketoolkit.preprocessing.**transform**(*recording*, *scalar=1*, *offset=0*)

Transforms the traces from the given recording extractor with a scalar and offset. New traces = traces*scalar + offset.

recording: RecordingExtractor The recording extractor to be transformed

scalar: float or array Scalar for the traces of the recording extractor or array with scalars for each channel

offset: float or array Offset for the traces of the recording extractor or array with offsets for each channel

transform_traces: TransformTracesRecording The transformed traces recording extractor object

spiketoolkit.preprocessing.**whiten**(*recording*, *chunk_size=30000*, *cache_chunks=False*, *seed=0*)

Whitens the recording extractor traces.

recording: RecordingExtractor The recording extractor to be whitened.

chunk_size: int The chunk size to be used for the filtering.

cache_chunks: bool If True, filtered traces are computed and cached all at once (default False).

seed: int Random seed for reproducibility

whitened_recording: WhitenRecording The whitened recording extractor

spiketoolkit.preprocessing.**common_reference**(*recording*, *reference='median'*, *groups=None*, *ref_channels=None*, *local_radius=(30, 55)*, *dtype=None*, *verbose=False*)

Re-references the recording extractor traces.

recording: RecordingExtractor The recording extractor to be re-referenced

reference: str 'median', 'average', 'single' or 'local' If 'median', common median reference (CMR) is implemented (the median of the selected channels is removed for each timestamp). If 'average', common average reference (CAR) is implemented (the mean of the selected channels is removed for each timestamp). If 'single', the selected channel(s) is remove from all channels. If 'local', an average CAR is implemented with only k channels selected the nearest outside of a radius around each channel

groups: list List of lists containing the channels for splitting the reference. The CMR, CAR, or referencing with respect to single channels are applied group-wise. However, this is not applied for the local CAR. It is useful when dealing with different channel groups, e.g. multiple tetrodes.

ref_channels: list or int If no 'groups' are specified, all channels are referenced to 'ref_channels'. If 'groups' is provided, then a list of channels to be applied to each group is expected. If 'single' reference, a list of one channel or an int is expected.

local_radius: tuple(int, int) Use in the local CAR implementation as the selecting annulus (exclude radius, include radius)

dtype: str dtype of the returned traces. If None, dtype is maintained

verbose: bool If True, output is verbose

referenced_recording: CommonReferenceRecording The re-referenced recording extractor object

9.2.2 Postprocessing

spiketoolkit.postprocessing.**get_unit_waveforms**(*recording*, *sorting*, *unit_ids=None*, *channel_ids=None*, *return_idxs=False*, *chunk_size=None*, *chunk_mb=500*, ***kwargs*)

Computes the spike waveforms from a recording and sorting extractor. The recording is split in chunks (the size in Mb is set with the chunk_mb argument) and all waveforms are extracted for each chunk and then re-assembled. If multiple jobs are used (n_jobs > 1), more and smaller chunks are created and processed in parallel.

recording: RecordingExtractor The recording extractor

sorting: SortingExtractor The sorting extractor

unit_ids: list List of unit ids to extract waveforms

channel_ids: list List of channels ids to compute waveforms from

return_idxs: bool If True, spike indexes and channel indexes are returned

chunk_size: int Size of chunks in number of samples. If None, it is automatically calculated

chunk_mb: int Size of chunks in Mb (default 500 Mb)

****kwargs: Keyword arguments** A dictionary with default values can be retrieved with: st.postprocessing.get_waveforms_params():

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned.

n_jobs: int Number of parallel jobs (default 1)

max_spikes_per_unit: int The maximum number of spikes to extract per unit.

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

seed: int Random seed for extracting random waveforms

save_property_or_features: bool If True (default), waveforms are saved as features of the sorting extractor object

recompute_info: bool If True, waveforms are recomputed (default False)

verbose: bool If True output is verbose

waveforms: list List of np.array (n_spikes, n_channels, n_timepoints) containing extracted waveforms for each unit

spike_indexes: list List of spike indexes for which waveforms are computed. Returned if 'return_idxs' is True

channel_indexes: list List of max channel indexes

spiketoolkit.postprocessing.**get_unit_templates**(*recording*, *sorting*, *unit_ids=None*, *channel_ids=None*, *mode='median'*, *_waveforms=None*, ***kwargs*)

Computes the spike templates from a recording and sorting extractor. If waveforms are not found as features, they are computed.

recording: RecordingExtractor The recording extractor

sorting: SortingExtractor The sorting extractor

unit_ids: list List of unit ids to extract templates

channel_ids: list List of channels ids to compute templates from

mode: str Use 'mean' or 'median' to compute templates

_waveforms: list Pre-computed waveforms to be used for computing templates

****kwargs: Keyword arguments** A dictionary with default values can be retrieved with: st.postprocessing.get_waveforms_params():

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned

n_jobs: int Number of parallel jobs (default 1)

max_spikes_per_unit: int The maximum number of spikes to extract per unit

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

seed: int Random seed for extracting random waveforms

save_property_or_features: bool If True (default), waveforms are saved as features of the sorting extractor object

recompute_info: bool If True, waveforms are recomputed (default False)

verbose: bool If True output is verbose

templates: list List of np.array (n_channels, n_timepoints) containing extracted templates for each unit

spiketoolkit.postprocessing.**get_unit_amplitudes**(*recording*, *sorting*, *unit_ids=None*, *channel_ids=None*, *return_idxs=False*, ***kwargs*)

Computes the spike amplitudes from a recording and sorting extractor. Amplitudes can be computed in absolute value (uV) or relative to the template amplitude.

recording: RecordingExtractor The recording extractor

sorting: SortingExtractor The sorting extractor

unit_ids: list List of unit ids to extract maximum channels

channel_ids: list List of channels ids to compute amplitudes from

return_idxs: bool If True, spike indexes and channel indexes are returned

****kwargs: Keyword arguments** A dictionary with default values can be retrieved with: st.postprocessing.get_waveforms_params():

method: str If 'absolute' (default), amplitudes are absolute amplitudes in uV are returned. If 'relative', amplitudes are returned as ratios between waveform amplitudes and template amplitudes.

peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

frames_before: int Frames before peak to compute amplitude

frames_after: float Frames after peak to compute amplitude

max_spikes_per_unit: int The maximum number of spikes to extract per unit

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

seed: int Random seed for extracting random waveforms

save_property_or_features: bool If True (default), waveforms are saved as features of the sorting extractor object

recompute_info: bool If True, waveforms are recomputed (default False)

n_jobs: int Number of jobs for parallelization. Default is None (no parallelization)

joblib_backend: str The backend for joblib. Default is 'loky'

verbose: bool If True output is verbose

amplitudes: list List of int containing extracted amplitudes for each unit

indexes: list List of spike indexes for which amplitudes are computed. Returned if 'return_idxs' is True

spiketoolkit.postprocessing.**get_unit_max_channels**(*recording*, *sorting*, *unit_ids=None*, *channel_ids=None*, *max_channels=1*, *peak='both'*, *mode='median'*, ***kwargs*)

Computes the spike maximum channels from a recording and sorting extractor. If templates are not found as property, they are computed. If templates are computed by group, the max channels refer to the overall channel ids.

recording: RecordingExtractor The recording extractor

sorting: SortingExtractor The sorting extractor

unit_ids: list List of unit ids to extract maximum channels

channel_ids: list List of channels ids to compute max_channels from

max_channels: int Number of max channels per units to return (default=1)

mode: str Use 'mean' or 'median' to compute templates

****kwargs: Keyword arguments** A dictionary with default values can be retrieved with: st.postprocessing.get_waveforms_params():

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned

n_jobs: int Number of parallel jobs (default 1)

max_spikes_per_unit: int The maximum number of spikes to extract per unit

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

seed: int Random seed for extracting random waveforms

save_property_or_features: bool If True (default), waveforms are saved as features of the sorting extractor object

recompute_info: bool If True, waveforms are recomputed (default False)

verbose: bool If True output is verbose

max_channels: list List of int containing extracted maximum channels for each unit

spiketoolkit.postprocessing.**set_unit_properties_by_max_channel_properties**(*recording*, *sorting*, *property*, *unit_ids=None*, *peak='both'*, *mode='median'*, *verbose=False*, ***kwargs*)

Extracts 'property' from recording channel with largest peak for each unit and saves it as unit property.

recording: RecordingExtractor The recording extractor

sorting: SortingExtractor The sorting extractor

property: str Property to compute

unit_ids: list List of unit ids to extract maximum channels

peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

mode: str Use 'mean' or 'median' to compute templates

verbose: bool If True output is verbose

****kwargs: Keyword arguments** A dictionary with default values can be retrieved with: st.postprocessing.get_waveforms_params():

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

max_spikes_per_unit: int The maximum number of spikes to extract per unit

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

seed: int Random seed for extracting random waveforms

n_jobs: int Number of parallel jobs (default 1)

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned

spiketoolkit.postprocessing.**compute_unit_pca_scores**(*recording*, *sorting*, *unit_ids=None*, *channel_ids=None*, *return_idxs=False*, *_waveforms=None*, *_spike_index_list=None*, *_channel_index_list=None*, ***kwargs*)

Computes the PCA scores from the unit waveforms. If waveforms are not found as features, they are computed.

recording: RecordingExtractor The recording extractor

sorting: SortingExtractor The sorting extractor

unit_ids: list List of unit ids to compute pca scores

channel_ids: list List of channels ids to compute pca from

return_idxs: list List of indexes of used spikes for each unit

_waveforms: list Pre-computed waveforms (optional)

_spike_index_list: list Pre-computed spike indexes for waveforms (optional)

_channel_index_list: list Pre-computed channel indexes for waveforms (optional)

****kwargs: Keyword arguments** A dictionary with default values can be retrieved with: st.postprocessing.get_waveforms_params():

n_comp: int Number of PCA components (default 3)

by_electrode: bool If True, PCA scores are computed electrode-wise (channel by channel)

max_spikes_for_pca: int The maximum number of spike per unit to use to fit the PCA.

whiten: bool If True, PCA is run with whiten equal True

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned

n_jobs: int Number of parallel jobs (default 1)

max_spikes_per_unit: int The maximum number of spikes to extract per unit

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

seed: int Random seed for extracting random waveforms

save_property_or_features: bool If True (default), waveforms are saved as features of the sorting extractor object

recompute_info: bool If True, waveforms are recomputed (default False)

verbose: bool If True output is verbose

pcs_scores: list List of np.array containing extracted pca scores. If 'by_electrode' is False, the array has shape (n_spikes, n_comp) If 'by_electrode' is True, the array has shape (n_spikes, n_channels, n_comp)

indexes: list List of spike indexes for which pca scores are computed. Returned if 'return_idxs' is True

spiketoolkit.postprocessing.**export_to_phy**(*recording*, *sorting*, *output_folder*, *compute_pc_features=True*, *compute_amplitudes=True*, *max_channels_per_template=16*, *copy_binary=True*, ***kwargs*)

Exports paired recording and sorting extractors to phy template-gui format.

recording: RecordingExtractor The recording extractor

sorting: SortingExtractor The sorting extractor

output_folder: str The output folder where the phy template-gui files are saved

compute_pc_features: bool If True (default), pc features are computed

compute_amplitudes: bool If True (default), waveforms amplitudes are compute

max_channels_per_template: int or None Maximum channels per unit to return. If None, all channels are returned

copy_binary: bool If True, the recording is copied and saved in the phy 'output_folder'. If False and the 'recording' is a CacheRecordingExtractor or a BinDatRecordingExtractor, then a relative link to the file recording location is used. Otherwise, the recording is not copied and the recording path is set to 'None'. (default True)

****kwargs: Keyword arguments** A dictionary with default values can be retrieved with: st.postprocessing.get_waveforms_params():

n_comp: int Number of PCA components (default 3)

max_spikes_for_pca: int The maximum number of spikes per unit to use to fit the PCA.

whiten: bool If True, PCA is run with whiten equal True

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

max_spikes_per_unit: int The maximum number of spikes to extract per unit

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

n_jobs: int Number of parallel jobs (default 1)

joblib_backend: str The backend for joblib. Default is 'loky'.

method: str If 'absolute' (default), amplitudes are absolute amplitudes in uV are returned. If 'relative', amplitudes are returned as ratios between waveform amplitudes and template amplitudes.

peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

frames_before: int Frames before peak to compute amplitude

frames_after: float Frames after peak to compute amplitude

recompute_info: bool If True, will always re-extract waveforms and templates.

save_property_or_features: bool If True, will store all calculated features and properties

verbose: bool If True output is verbose

seed: int Random seed for extracting random waveforms

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

filter_flag: bool If False, will not display the warning on non-filtered recording. Default is True.

spiketoolkit.postprocessing.**compute_unit_template_features**(*recording*, *sorting*, *unit_ids=None*, *channel_ids=None*, *feature_names=None*, *max_channels_per_features=1*, *recovery_slope_window=0.7*, *upsampling_factor=1*, *invert_waveforms=False*, *as_dataframe=False*, ***kwargs*)

Use SpikeInterface/spikefeatures to compute features for the unit template.

These consist of a set of 1D features:

- peak to valley (peak_to_valley), time between peak and valley
- halfwidth (halfwidth), width of peak at half its amplitude
- peak trough ratio (peak_trough_ratio), amplitude of peak over amplitude of trough
- repolarization slope (repolarization_slope), slope between trough and return to base
- recovery slope (recovery_slope), slope after peak towards baseline

And 2D features:

- unit_spread
- propagation velocity

To be implemented

The metrics are computed on 'negative' waveforms, if templates are saved as positive, pass keyword 'invert_waveforms'.

recording: RecordingExtractor The recording extractor

sorting: SortingExtractor The sorting extractor

unit_ids: list List of unit ids to compute features

channel_ids: list List of channels ids to compute templates on which features are computed

feature_names: list List of feature names to be computed. If None, all features are computed

max_channels_per_features: int Maximum number of channels to compute features on (default 1). If channel_ids is used, this parameter is ignored

upsampling_factor: int Factor with which to upsample the template resolution (default 1)

invert_waveforms: bool Invert templates before computing features (default False)

recovery_slope_window: float Window after peak in ms wherein to compute recovery slope (default 0.7)

as_dataframe: bool IfTrue, output is returned as a pandas dataframe, otherwise as a dictionary

****kwargs: Keyword arguments** A dictionary with default values can be retrieved with: st.postprocessing.get_waveforms_params():

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned

n_jobs: int Number of parallel jobs (default 1)

max_spikes_per_unit: int The maximum number of spikes to extract per unit

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

seed: int Random seed for extracting random waveforms

save_property_or_features: bool If True (default), waveforms are saved as features of the sorting extractor object

recompute_info: bool If True, waveforms are recomputed (default False)

verbose: bool If True output is verbose

features: dict or pandas.DataFrame The computed features as a dictionary or a pandas.DataFrame (if as_dataframe is True)

9.2.3 Validation

spiketoolkit.validation.**compute_isolation_distances**(*sorting*, *recording*, *num_channels_to_compare=13*, *max_spikes_per_cluster=500*, *unit_ids=None*, ***kwargs*)

Computes and returns the isolation distances in the sorted dataset.

sorting: SortingExtractor The sorting result to be evaluated.

recording: RecordingExtractor The given recording extractor from which to extract amplitudes

num_channels_to_compare: int The number of channels to be used for the PC extraction and comparison

max_spikes_per_cluster: int Max spikes to be used from each unit

unit_ids: list List of unit ids to compute metric for. If not specified, all units are used

****kwargs: keyword arguments**

Keyword arguments among the following:

method: str If 'absolute' (default), amplitudes are absolute amplitudes in uV are returned. If 'relative', amplitudes are returned as ratios between waveform amplitudes and template amplitudes

peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

frames_before: int Frames before peak to compute amplitude

frames_after: int Frames after peak to compute amplitude

apply_filter: bool If True, recording is bandpass-filtered

freq_min: float High-pass frequency for optional filter (default 300 Hz)

freq_max: float Low-pass frequency for optional filter (default 6000 Hz)

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned

n_jobs: int Number of parallel jobs (default 1)

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

save_property_or_features: bool If true, it will save features in the sorting extractor

recompute_info: bool If True, waveforms are recomputed

max_spikes_per_unit: int The maximum number of spikes to extract per unit

seed: int Random seed for reproducibility

verbose: bool If True, will be verbose in metric computation

isolation_distances: np.ndarray The isolation distances of the sorted units.

spiketoolkit.validation.**compute_isi_violations**(*sorting*, *duration_in_frames*, *isi_threshold=0.0015*, *min_isi=None*, *sampling_frequency=None*, *unit_ids=None*, ***kwargs*)

Computes and returns the isi violations for the sorted dataset.

sorting: SortingExtractor The sorting result to be evaluated.

duration_in_frames: int Length of recording (in frames).

isi_threshold: float The isi threshold for calculating isi violations

min_isi: float The minimum expected isi value

sampling_frequency: float The sampling frequency of the result. If None, will check to see if sampling frequency is in sorting extractor

unit_ids: list List of unit ids to compute metric for. If not specified, all units are used

****kwargs: keyword arguments**

Keyword arguments among the following:

save_property_or_features: bool If True, the metric is saved as sorting property

verbose: bool If True, will be verbose in metric computation

isi_violations: np.ndarray The isi violations of the sorted units.

spiketoolkit.validation.**compute_snrs**(*sorting*, *recording*, *snr_mode='mad'*, *snr_noise_duration=10.0*, *max_spikes_per_unit_for_snr=1000*, *template_mode='median'*, *max_channel_peak='both'*, *unit_ids=None*, ***kwargs*)

Computes and returns the snrs in the sorted dataset.

sorting: SortingExtractor The sorting result to be evaluated.

recording: RecordingExtractor The given recording extractor from which to extract amplitudes

snr_mode: str Mode to compute noise SNR ('mad' | 'std' - default 'mad')

snr_noise_duration: float Number of seconds to compute noise level from (default 10.0)

max_spikes_per_unit_for_snr: int Maximum number of spikes to compute templates from (default 1000)

template_mode: str Use 'mean' or 'median' to compute templates

max_channel_peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

unit_ids: list List of unit ids to compute metric for. If not specified, all units are used

****kwargs: keyword arguments**

Keyword arguments among the following:

method: str If 'absolute' (default), amplitudes are absolute amplitudes in uV are returned. If 'relative', amplitudes are returned as ratios between waveform amplitudes and template amplitudes

peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

frames_before: int Frames before peak to compute amplitude

frames_after: int Frames after peak to compute amplitude

apply_filter: bool If True, recording is bandpass-filtered

freq_min: float High-pass frequency for optional filter (default 300 Hz)

freq_max: float Low-pass frequency for optional filter (default 6000 Hz)

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned

n_jobs: int Number of parallel jobs (default 1)

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

save_property_or_features: bool If true, it will save features in the sorting extractor

recompute_info: bool If True, waveforms are recomputed

max_spikes_per_unit: int The maximum number of spikes to extract per unit

seed: int Random seed for reproducibility

verbose: bool If True, will be verbose in metric computation

snrs: np.ndarray The snrs of the sorted units.

spiketoolkit.validation.**compute_amplitude_cutoffs**(*sorting*, *recording*, *unit_ids=None*, ***kwargs*)

Computes and returns the amplitude cutoffs for the sorted dataset.

sorting: SortingExtractor The sorting result to be evaluated.

recording: RecordingExtractor The given recording extractor from which to extract amplitudes

unit_ids: list List of unit ids to compute metric for. If not specified, all units are used

****kwargs: keyword arguments**

Keyword arguments among the following:

apply_filter: bool If True, recording is bandpass-filtered.

freq_min: float High-pass frequency for optional filter (default 300 Hz)

freq_max: float Low-pass frequency for optional filter (default 6000 Hz)

save_property_or_features: bool If true, it will save amplitudes in the sorting extractor

recompute_info: bool If True, waveforms are recomputed

max_spikes_per_unit: int The maximum number of spikes to extract per unit

method: str If 'absolute' (default), amplitudes are absolute amplitudes in uV are returned. If 'relative', amplitudes are returned as ratios between waveform amplitudes and template amplitudes.

peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

frames_before: int Frames before peak to compute amplitude

frames_after: float Frames after peak to compute amplitude

save_property_or_features: bool If True, the metric is saved as sorting property

seed: int Random seed for reproducibility

verbose: bool If True, will be verbose in metric computation

amplitude_cutoffs: np.ndarray The amplitude cutoffs of the sorted units.

spiketoolkit.validation.**compute_d_primes**(*sorting*, *recording*, *num_channels_to_compare=13*, *max_spikes_per_cluster=500*, *unit_ids=None*, ***kwargs*)

Computes and returns the d primes in the sorted dataset.

sorting: SortingExtractor The sorting result to be evaluated

recording: RecordingExtractor The given recording extractor from which to extract amplitudes

num_channels_to_compare: int The number of channels to be used for the PC extraction and comparison

max_spikes_per_cluster: int Max spikes to be used from each unit

unit_ids: list List of unit ids to compute metric for. If not specified, all units are used

****kwargs: keyword arguments**

Keyword arguments among the following:

method: str If 'absolute' (default), amplitudes are absolute amplitudes in uV are returned. If 'relative', amplitudes are returned as ratios between waveform amplitudes and template amplitudes

peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

frames_before: int Frames before peak to compute amplitude

frames_after: int Frames after peak to compute amplitude

apply_filter: bool If True, recording is bandpass-filtered

freq_min: float High-pass frequency for optional filter (default 300 Hz)

freq_max: float Low-pass frequency for optional filter (default 6000 Hz)

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned

n_jobs: int Number of parallel jobs (default 1)

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

save_property_or_features: bool If true, it will save features in the sorting extractor

recompute_info: bool If True, waveforms are recomputed

max_spikes_per_unit: int The maximum number of spikes to extract per unit

seed: int Random seed for reproducibility

verbose: bool If True, will be verbose in metric computation

d_primes: np.ndarray The d primes of the sorted units.

spiketoolkit.validation.**compute_drift_metrics**(*sorting*, *recording*, *drift_metrics_interval_s=51*, *drift_metrics_min_spikes_per_interval=10*, *unit_ids=None*, ***kwargs*)

Computes and returns the drift metrics in the sorted dataset.

sorting: SortingExtractor The sorting result to be evaluated.

recording: RecordingExtractor The given recording extractor from which to extract amplitudes

drift_metrics_interval_s: float Time period for evaluating drift.

drift_metrics_min_spikes_per_interval: int Minimum number of spikes for evaluating drift metrics per interval.

unit_ids: list List of unit ids to compute metric for. If not specified, all units are used

****kwargs: keyword arguments**

Keyword arguments among the following:

method: str If 'absolute' (default), amplitudes are absolute amplitudes in uV are returned. If 'relative', amplitudes are returned as ratios between waveform amplitudes and template amplitudes

peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

frames_before: int Frames before peak to compute amplitude

frames_after: int Frames after peak to compute amplitude

apply_filter: bool If True, recording is bandpass-filtered

freq_min: float High-pass frequency for optional filter (default 300 Hz)

freq_max: float Low-pass frequency for optional filter (default 6000 Hz)

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned

n_jobs: int Number of parallel jobs (default 1)

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

save_property_or_features: bool If true, it will save features in the sorting extractor

recompute_info: bool If True, waveforms are recomputed

max_spikes_per_unit: int The maximum number of spikes to extract per unit

seed: int Random seed for reproducibility

verbose: bool If True, will be verbose in metric computation

dm_metrics: np.ndarray The drift metrics of the sorted units.

spiketoolkit.validation.**compute_firing_rates**(*sorting*, *duration_in_frames*, *sampling_frequency=None*, *unit_ids=None*, ***kwargs*)

Computes and returns the firing rates for the sorted dataset.

sorting: SortingExtractor The sorting result to be evaluated.

duration_in_frames: int Length of recording (in frames).

sampling_frequency: float The sampling frequency of the result. If None, will check to see if sampling frequency is in sorting extractor

unit_ids: list List of unit ids to compute metric for. If not specified, all units are used

****kwargs: keyword arguments**

Keyword arguments among the following:

save_property_or_features: bool If True, the metric is saved as sorting property

verbose: bool If True, will be verbose in metric computation

firing_rates: np.ndarray The firing rates of the sorted units.

spiketoolkit.validation.**compute_l_ratios**(*sorting*, *recording*, *num_channels_to_compare=13*, *max_spikes_per_cluster=500*, *unit_ids=None*, ***kwargs*)

Computes and returns the l ratios in the sorted dataset.

sorting: SortingExtractor The sorting result to be evaluated

recording: RecordingExtractor The given recording extractor from which to extract amplitudes

num_channels_to_compare: int The number of channels to be used for the PC extraction and comparison

max_spikes_per_cluster: int Max spikes to be used from each unit

unit_ids: list List of unit ids to compute metric for. If not specified, all units are used

****kwargs: keyword arguments**

Keyword arguments among the following:

method: str If 'absolute' (default), amplitudes are absolute amplitudes in uV are returned. If 'relative', amplitudes are returned as ratios between waveform amplitudes and template amplitudes

peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

frames_before: int Frames before peak to compute amplitude

frames_after: int Frames after peak to compute amplitude

apply_filter: bool If True, recording is bandpass-filtered

freq_min: float High-pass frequency for optional filter (default 300 Hz)

freq_max: float Low-pass frequency for optional filter (default 6000 Hz)

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned

n_jobs: int Number of parallel jobs (default 1)

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

save_property_or_features: bool If true, it will save features in the sorting extractor

recompute_info: bool If True, waveforms are recomputed

max_spikes_per_unit: int The maximum number of spikes to extract per unit

seed: int Random seed for reproducibility

verbose: bool If True, will be verbose in metric computation

l_ratios: np.ndarray The l ratios of the sorted units.

spiketoolkit.validation.**compute_nn_metrics**(*sorting*, *recording*, *num_channels_to_compare=13*, *max_spikes_per_cluster=500*, *max_spikes_for_nn=10000*, *n_neighbors=4*, *unit_ids=None*, ***kwargs*)

Computes and returns the nearest neighbor metrics in the sorted dataset.

sorting: SortingExtractor The sorting result to be evaluated.

recording: RecordingExtractor The given recording extractor from which to extract amplitudes

num_channels_to_compare: int The number of channels to be used for the PC extraction and comparison

max_spikes_per_cluster: int Max spikes to be used from each unit

max_spikes_for_nn: int Max spikes to be used for nearest-neighbors calculation

n_neighbors: int Number of neighbors to compare

unit_ids: list List of unit ids to compute metric for. If not specified, all units are used

****kwargs: keyword arguments**

Keyword arguments among the following:

method: str If 'absolute' (default), amplitudes are absolute amplitudes in uV are returned. If 'relative', amplitudes are returned as ratios between waveform amplitudes and template amplitudes

peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

frames_before: int Frames before peak to compute amplitude

frames_after: int Frames after peak to compute amplitude

apply_filter: bool If True, recording is bandpass-filtered

freq_min: float High-pass frequency for optional filter (default 300 Hz)

freq_max: float Low-pass frequency for optional filter (default 6000 Hz)

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned

n_jobs: int Number of parallel jobs (default 1)

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

save_property_or_features: bool If true, it will save features in the sorting extractor

recompute_info: bool If True, waveforms are recomputed

max_spikes_per_unit: int The maximum number of spikes to extract per unit

seed: int Random seed for reproducibility

verbose: bool If True, will be verbose in metric computation

nn_metrics: np.ndarray The nearest neighbor metrics of the sorted units.

spiketoolkit.validation.**compute_num_spikes**(*sorting*, *sampling_frequency=None*, *unit_ids=None*, ***kwargs*)

Computes and returns the num spikes for the sorted dataset.

sorting: SortingExtractor The sorting result to be evaluated

sampling_frequency: float The sampling frequency of the result. If None, will check to see if sampling frequency is in sorting extractor

unit_ids: list List of unit ids to compute metric for. If not specified, all units are used

****kwargs: keyword arguments**

Keyword arguments among the following:

save_property_or_features: bool If True, the metric is saved as sorting property

verbose: bool If True, will be verbose in metric computation

num_spikes: np.ndarray The number of spikes of the sorted units.

spiketoolkit.validation.**compute_presence_ratios**(*sorting*, *duration_in_frames*, *sampling_frequency=None*, *unit_ids=None*, ***kwargs*)

Computes and returns the presence ratios for the sorted dataset.

sorting: SortingExtractor The sorting result to be evaluated.

duration_in_frames: int Length of recording (in frames).

sampling_frequency: float The sampling frequency of the result. If None, will check to see if sampling frequency is in sorting extractor

unit_ids: list List of unit ids to compute metric for. If not specified, all units are used

****kwargs: keyword arguments**

Keyword arguments among the following:

save_property_or_features: bool If True, the metric is saved as sorting property

verbose: bool If True, will be verbose in metric computation

presence_ratios: np.ndarray The presence ratios of the sorted units.

spiketoolkit.validation.**compute_silhouette_scores**(*sorting*, *recording*, *max_spikes_for_silhouette=10000*, *unit_ids=None*, ***kwargs*)

Computes and returns the silhouette scores in the sorted dataset.

sorting: SortingExtractor The sorting result to be evaluated

recording: RecordingExtractor The given recording extractor from which to extract amplitudes

max_spikes_for_silhouette: int Max spikes to be used for silhouette metric

unit_ids: list List of unit ids to compute metric for. If not specified, all units are used

****kwargs: keyword arguments**

Keyword arguments among the following:

method: str If 'absolute' (default), amplitudes are absolute amplitudes in uV are returned. If 'relative', amplitudes are returned as ratios between waveform amplitudes and template amplitudes

peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

frames_before: int Frames before peak to compute amplitude

frames_after: int Frames after peak to compute amplitude

apply_filter: bool If True, recording is bandpass-filtered

freq_min: float High-pass frequency for optional filter (default 300 Hz)

freq_max: float Low-pass frequency for optional filter (default 6000 Hz)

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned

n_jobs: int Number of parallel jobs (default 1)

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

save_property_or_features: bool If true, it will save features in the sorting extractor

recompute_info: bool If True, waveforms are recomputed

max_spikes_per_unit: int The maximum number of spikes to extract per unit

seed: int Random seed for reproducibility

verbose: bool If True, will be verbose in metric computation

silhouette_scores: np.ndarray The sihouette scores of the sorted units.

spiketoolkit.validation.**compute_quality_metrics**(*sorting*, *recording=None*, *duration_in_frames=None*, *sampling_frequency=None*, *metric_names=None*, *unit_ids=None*, *as_dataframe=False*, *isi_threshold=0.0015*, *min_isi=None*, *snr_mode='mad'*, *snr_noise_duration=10.0*, *max_spikes_per_unit_for_snr=1000*, *template_mode='median'*, *max_channel_peak='both'*, *max_spikes_per_unit_for_noise_overlap=1000*, *noise_overlap_num_features=10*, *noise_overlap_num_knn=6*, *drift_metrics_interval_s=51*, *drift_metrics_min_spikes_per_interval=10*, *max_spikes_for_silhouette=10000*, *num_channels_to_compare=13*, *max_spikes_per_cluster=500*, *max_spikes_for_nn=10000*, *n_neighbors=4*, ***kwargs*)

Computes and returns all specified metrics for the sorted dataset.

sorting: SortingExtractor The sorting result to be evaluated.

recording: RecordingExtractor The given recording extractor from which to extract amplitudes

duration_in_frames: int Length of recording (in frames).

sampling_frequency: float The sampling frequency of the result. If None, will check to see if sampling frequency is in sorting extractor

metric_names: list List of metric names to be computed

unit_ids: list List of unit ids to compute metric for. If not specified, all units are used

as_dataframe: bool If True, will return dataframe of metrics. If False, will return dictionary.

isi_threshold: float The isi threshold for calculating isi violations

min_isi: float The minimum expected isi value

snr_mode: str Mode to compute noise SNR ('mad' | 'std' - default 'mad')

snr_noise_duration: float Number of seconds to compute noise level from (default 10.0)

max_spikes_per_unit_for_snr: int Maximum number of spikes to compute templates for SNR from (default 1000)

template_mode: str Use 'mean' or 'median' to compute templates

max_channel_peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

max_spikes_per_unit_for_noise_overlap: int Maximum number of spikes to compute templates for noise overlap from (default 1000)

noise_overlap_num_features: int Number of features to use for PCA for noise overlap

noise_overlap_num_knn: int Number of nearest neighbors for noise overlap

drift_metrics_interval_s: float Time period for evaluating drift.

drift_metrics_min_spikes_per_interval: int Minimum number of spikes for evaluating drift metrics per interval

max_spikes_for_silhouette: int Max spikes to be used for silhouette metric

num_channels_to_compare: int The number of channels to be used for the PC extraction and comparison

max_spikes_per_cluster: int Max spikes to be used from each unit

max_spikes_for_nn: int Max spikes to be used for nearest-neighbors calculation

n_neighbors: int Number of neighbors to compare

****kwargs: keyword arguments**

Keyword arguments among the following:

method: str If 'absolute' (default), amplitudes are absolute amplitudes in uV are returned. If 'relative', amplitudes are returned as ratios between waveform amplitudes and template amplitudes

peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

frames_before: int Frames before peak to compute amplitude

frames_after: int Frames after peak to compute amplitude

apply_filter: bool If True, recording is bandpass-filtered

freq_min: float High-pass frequency for optional filter (default 300 Hz)

freq_max: float Low-pass frequency for optional filter (default 6000 Hz)

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned

n_jobs: int Number of parallel jobs (default 1)

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

save_property_or_features: bool If true, it will save features in the sorting extractor

recompute_info: bool If True, waveforms are recomputed

max_spikes_per_unit: int The maximum number of spikes to extract per unit

seed: int Random seed for reproducibility

verbose: bool If True, will be verbose in metric computation

metrics: dictionary OR pandas.dataframe Dictionary or pandas.dataframe of metrics.

9.2.4 Curation

spiketoolkit.curation.**threshold_amplitude_cutoffs**(*sorting*, *recording*, *threshold*, *threshold_sign*, ***kwargs*)

Computes and thresholds the amplitude cutoffs in the sorted dataset with the given sign and value.

sorting: SortingExtractor The sorting result to be evaluated

recording: RecordingExtractor The given recording extractor

threshold: int or float The threshold for the given metric

threshold_sign: str If 'less', will threshold any metric less than the given threshold If 'less_or_equal', will threshold any metric less than or equal to the given threshold If 'greater', will threshold any metric greater than the given threshold If 'greater_or_equal', will threshold any metric greater than or equal to the given threshold

****kwargs: keyword arguments**

Keyword arguments among the following:

method: str If 'absolute' (default), amplitudes are absolute amplitudes in uV are returned If 'relative', amplitudes are returned as ratios between waveform amplitudes and template amplitudes

peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

frames_before: int Frames before peak to compute amplitude

frames_after: int Frames after peak to compute amplitude

apply_filter: bool If True, recording is bandpass-filtered

freq_min: float High-pass frequency for optional filter (default 300 Hz)

freq_max: float Low-pass frequency for optional filter (default 6000 Hz)

save_property_or_features: bool If true, it will save features in the sorting extractor

recompute_info: bool If True, waveforms are recomputed

max_spikes_per_unit: int The maximum number of spikes to extract per unit

threshold sorting extractor

spiketoolkit.curation.**threshold_d_primes**(*sorting*, *recording*, *threshold*, *threshold_sign*, *num_channels_to_compare=13*, *max_spikes_per_cluster=500*, ***kwargs*)

Computes and thresholds the d primes in the sorted dataset with the given sign and value.

sorting: SortingExtractor The sorting result to be evaluated

recording: RecordingExtractor The given recording extractor

threshold: int or float The threshold for the given metric

threshold_sign: str If 'less', will threshold any metric less than the given threshold If 'less_or_equal', will threshold any metric less than or equal to the given threshold If 'greater', will threshold any metric greater than the given threshold If 'greater_or_equal', will threshold any metric greater than or equal to the given threshold

num_channels_to_compare: int The number of channels to be used for the PC extraction and comparison

max_spikes_per_cluster: int Max spikes to be used from each unit

****kwargs: keyword arguments**

Keyword arguments among the following:

method: str If 'absolute' (default), amplitudes are absolute amplitudes in uV are returned. If 'relative', amplitudes are returned as ratios between waveform amplitudes and template amplitudes

peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

frames_before: int Frames before peak to compute amplitude

frames_after: int Frames after peak to compute amplitude

apply_filter: bool If True, recording is bandpass-filtered

freq_min: float High-pass frequency for optional filter (default 300 Hz)

freq_max: float Low-pass frequency for optional filter (default 6000 Hz)

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned

n_jobs: int Number of parallel jobs (default 1)

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

save_property_or_features: bool If true, it will save features in the sorting extractor

recompute_info: bool If True, waveforms are recomputed

max_spikes_per_unit: int The maximum number of spikes to extract per unit

seed: int Random seed for reproducibility

verbose: bool If True, will be verbose in metric computation

threshold sorting extractor

spiketoolkit.curation.**threshold_drift_metrics**(*sorting*, *recording*, *threshold*, *threshold_sign*, *metric_name='max_drift'*, *drift_metrics_interval_s=51*, *drift_metrics_min_spikes_per_interval=10*, ***kwargs*)

Computes and thresholds the specified drift metric for the sorted dataset with the given sign and value.

sorting: SortingExtractor The sorting result to be evaluated.

recording: RecordingExtractor The given recording extractor

threshold: int or float The threshold for the given metric.

threshold_sign: str If 'less', will threshold any metric less than the given threshold. If 'less_or_equal', will threshold any metric less than or equal to the given threshold. If 'greater', will threshold any metric greater than the given threshold. If 'greater_or_equal', will threshold any metric greater than or equal to the given threshold.

metric_name: str The name of the drift metric to be thresholded (either "max_drift" or "cumulative_drift").

drift_metrics_interval_s: float Time period for evaluating drift.

drift_metrics_min_spikes_per_interval: int Minimum number of spikes for evaluating drift metrics per interval.

****kwargs: keyword arguments**

Keyword arguments among the following:

method: str If 'absolute' (default), amplitudes are absolute amplitudes in uV are returned. If 'relative', amplitudes are returned as ratios between waveform amplitudes and template amplitudes

peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

frames_before: int Frames before peak to compute amplitude

frames_after: int Frames after peak to compute amplitude

apply_filter: bool If True, recording is bandpass-filtered

freq_min: float High-pass frequency for optional filter (default 300 Hz)

freq_max: float Low-pass frequency for optional filter (default 6000 Hz)

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned

n_jobs: int Number of parallel jobs (default 1)

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

save_property_or_features: bool If true, it will save features in the sorting extractor

recompute_info: bool If True, waveforms are recomputed

max_spikes_per_unit: int The maximum number of spikes to extract per unit

seed: int Random seed for reproducibility

verbose: bool If True, will be verbose in metric computation

threshold sorting extractor

spiketoolkit.curation.**threshold_firing_rates**(*sorting*, *threshold*, *threshold_sign*, *duration_in_frames*, *sampling_frequency=None*, ***kwargs*)

Computes and thresholds the firing rates in the sorted dataset with the given sign and value.

sorting: SortingExtractor The sorting result to be evaluated

threshold: int or float The threshold for the given metric

threshold_sign: str If 'less', will threshold any metric less than the given threshold If 'less_or_equal', will threshold any metric less than or equal to the given threshold If 'greater', will threshold any metric greater than the given threshold If 'greater_or_equal', will threshold any metric greater than or equal to the given threshold

duration_in_frames: int Length of recording (in frames).

sampling_frequency: The sampling frequency of the result. If None, will check to see if sampling frequency is in sorting extractor

****kwargs: keyword arguments**

Keyword arguments among the following:

save_property_or_features: bool If True, the metric is saved as sorting property

verbose: bool If True, will be verbose in metric computation

threshold sorting extractor

spiketoolkit.curation.**threshold_isi_violations**(*sorting*, *threshold*, *threshold_sign*, *duration_in_frames*, *isi_threshold=0.0015*, *min_isi=None*, *sampling_frequency=None*, ***kwargs*)

Computes and thresholds the isi violations in the sorted dataset with the given sign and value.

sorting: SortingExtractor The sorting result to be evaluated.

threshold: int or float The threshold for the given metric.

threshold_sign: str If 'less', will threshold any metric less than the given threshold If 'less_or_equal', will threshold any metric less than or equal to the given threshold If 'greater', will threshold any metric greater than the given threshold If 'greater_or_equal', will threshold any metric greater than or equal to the given threshold

duration_in_frames: int Length of recording (in frames).

isi_threshold: float The isi threshold for calculating isi violations.

min_isi: float The minimum expected isi value.

sampling_frequency: The sampling frequency of the result. If None, will check to see if sampling frequency is in sorting extractor.

****kwargs: keyword arguments**

Keyword arguments among the following:

save_property_or_features: bool If True, the metric is saved as sorting property

verbose: bool If True, will be verbose in metric computation

threshold sorting extractor

spiketoolkit.curation.**threshold_isolation_distances**(*sorting*, *recording*, *threshold*, *threshold_sign*, *num_channels_to_compare=13*, *max_spikes_per_cluster=500*, ***kwargs*)

Computes and thresholds the isolation distances in the sorted dataset with the given sign and value.

sorting: SortingExtractor The sorting result to be evaluated.

recording: RecordingExtractor The given recording extractor

threshold: int or float The threshold for the given metric.

threshold_sign: str If 'less', will threshold any metric less than the given threshold. If 'less_or_equal', will threshold any metric less than or equal to the given threshold. If 'greater', will threshold any metric greater than the given threshold. If 'greater_or_equal', will threshold any metric greater than or equal to the given threshold.

num_channels_to_compare: int The number of channels to be used for the PC extraction and comparison

max_spikes_per_cluster: int Max spikes to be used from each unit

****kwargs: keyword arguments**

Keyword arguments among the following:

method: str If 'absolute' (default), amplitudes are absolute amplitudes in uV are returned. If 'relative', amplitudes are returned as ratios between waveform amplitudes and template amplitudes

peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

frames_before: int Frames before peak to compute amplitude

frames_after: int Frames after peak to compute amplitude

apply_filter: bool If True, recording is bandpass-filtered

freq_min: float High-pass frequency for optional filter (default 300 Hz)

freq_max: float Low-pass frequency for optional filter (default 6000 Hz)

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned

n_jobs: int Number of parallel jobs (default None)

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

save_property_or_features: bool If true, it will save features in the sorting extractor

recompute_info: bool If True, waveforms are recomputed

max_spikes_per_unit: int The maximum number of spikes to extract per unit

seed: int Random seed for reproducibility

verbose: bool If True, will be verbose in metric computation

threshold sorting extractor

spiketoolkit.curation.**threshold_l_ratios**(*sorting*, *recording*, *threshold*, *threshold_sign*, *num_channels_to_compare=13*, *max_spikes_per_cluster=500*, ***kwargs*)

Computes and thresholds the l ratios in the sorted dataset with the given sign and value.

sorting: SortingExtractor The sorting result to be evaluated.

recording: RecordingExtractor The given recording extractor

threshold: int or float The threshold for the given metric.

threshold_sign: str If 'less', will threshold any metric less than the given threshold. If 'less_or_equal', will threshold any metric less than or equal to the given threshold. If 'greater', will threshold any metric greater than the given threshold. If 'greater_or_equal', will threshold any metric greater than or equal to the given threshold.

num_channels_to_compare: int The number of channels to be used for the PC extraction and comparison

max_spikes_per_cluster: int Max spikes to be used from each unit

****kwargs: keyword arguments**

Keyword arguments among the following:

method: str If 'absolute' (default), amplitudes are absolute amplitudes in uV are returned. If 'relative', amplitudes are returned as ratios between waveform amplitudes and template amplitudes

peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

frames_before: int Frames before peak to compute amplitude

frames_after: int Frames after peak to compute amplitude

apply_filter: bool If True, recording is bandpass-filtered

freq_min: float High-pass frequency for optional filter (default 300 Hz)

freq_max: float Low-pass frequency for optional filter (default 6000 Hz)

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned

n_jobs: int Number of parallel jobs (default 1)

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

save_property_or_features: bool If true, it will save features in the sorting extractor

recompute_info: bool If True, waveforms are recomputed

max_spikes_per_unit: int The maximum number of spikes to extract per unit

seed: int Random seed for reproducibility

verbose: bool If True, will be verbose in metric computation

threshold sorting extractor

spiketoolkit.curation.**threshold_nn_metrics**(*sorting*, *recording*, *threshold*, *threshold_sign*, *metric_name='nn_hit_rate'*, *num_channels_to_compare=13*, *max_spikes_per_cluster=500*, *max_spikes_for_nn=10000*, *n_neighbors=4*, ***kwargs*)

Computes and thresholds the specified nearest neighbor metric for the sorted dataset with the given sign and value.

sorting: SortingExtractor The sorting result to be evaluated.

recording: RecordingExtractor The given recording extractor

threshold: int or float The threshold for the given metric.

threshold_sign: str If 'less', will threshold any metric less than the given threshold. If 'less_or_equal', will threshold any metric less than or equal to the given threshold. If 'greater', will threshold any metric greater than the given threshold. If 'greater_or_equal', will threshold any metric greater than or equal to the given threshold.

metric_name: str The name of the nearest neighbor metric to be thresholded (either "nn_hit_rate" or "nn_miss_rate").

num_channels_to_compare: int The number of channels to be used for the PC extraction and comparison

max_spikes_per_cluster: int Max spikes to be used from each unit

max_spikes_for_nn: int Max spikes to be used for nearest-neighbors calculation.

n_neighbors: int Number of neighbors to compare.

****kwargs: keyword arguments**

Keyword arguments among the following:

method: str If 'absolute' (default), amplitudes are absolute amplitudes in uV are returned. If 'relative', amplitudes are returned as ratios between waveform amplitudes and template amplitudes

peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

frames_before: int Frames before peak to compute amplitude

frames_after: int Frames after peak to compute amplitude

apply_filter: bool If True, recording is bandpass-filtered

freq_min: float High-pass frequency for optional filter (default 300 Hz)

freq_max: float Low-pass frequency for optional filter (default 6000 Hz)

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned

n_jobs: int Number of parallel jobs (default 1)

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

save_property_or_features: bool If true, it will save features in the sorting extractor

recompute_info: bool If True, waveforms are recomputed

max_spikes_per_unit: int The maximum number of spikes to extract per unit

seed: int Random seed for reproducibility

verbose: bool If True, will be verbose in metric computation

threshold sorting extractor

spiketoolkit.curation.**threshold_num_spikes**(*sorting*, *threshold*, *threshold_sign*, *sampling_frequency=None*, ***kwargs*)

Computes and thresholds the num spikes in the sorted dataset with the given sign and value.

sorting: SortingExtractor The sorting result to be evaluated

threshold: int or float The threshold for the given metric

threshold_sign: str If 'less', will threshold any metric less than the given threshold If 'less_or_equal', will threshold any metric less than or equal to the given threshold If 'greater', will threshold any metric greater than the given threshold If 'greater_or_equal', will threshold any metric greater than or equal to the given threshold

sampling_frequency: float The sampling frequency of the result. If None, will check to see if sampling frequency is in sorting extractor

****kwargs: keyword arguments**

Keyword arguments among the following:

save_property_or_features: bool If True, the metric is saved as sorting property

verbose: bool If True, will be verbose in metric computation

threshold sorting extractor

spiketoolkit.curation.**threshold_presence_ratios**(*sorting*, *threshold*, *threshold_sign*, *duration_in_frames*, *sampling_frequency=None*, ***kwargs*)

Computes and thresholds the presence ratios in the sorted dataset with the given sign and value.

sorting: SortingExtractor The sorting result to be evaluated

threshold: int or float The threshold for the given metric

threshold_sign: str If 'less', will threshold any metric less than the given threshold If 'less_or_equal', will threshold any metric less than or equal to the given threshold If 'greater', will threshold any metric greater than the given threshold If 'greater_or_equal', will threshold any metric greater than or equal to the given threshold

duration_in_frames: int Length of recording (in frames).

sampling_frequency: The sampling frequency of the result. If None, will check to see if sampling frequency is in sorting extractor

****kwargs: keyword arguments**

Keyword arguments among the following:

save_property_or_features: bool If True, the metric is saved as sorting property

verbose: bool If True, will be verbose in metric computation

threshold sorting extractor

spiketoolkit.curation.**threshold_silhouette_scores**(*sorting*, *recording*, *threshold*, *threshold_sign*, *max_spikes_for_silhouette=10000*, ***kwargs*)

Computes and thresholds the silhouette scores in the sorted dataset with the given sign and value.

sorting: SortingExtractor The sorting result to be evaluated

recording: RecordingExtractor The given recording extractor

threshold: int or float The threshold for the given metric

threshold_sign: str If 'less', will threshold any metric less than the given threshold If 'less_or_equal', will threshold any metric less than or equal to the given threshold If 'greater', will threshold any metric greater than the given threshold If 'greater_or_equal', will threshold any metric greater than or equal to the given threshold

max_spikes_for_silhouette: int Max spikes to be used for silhouette metric

****kwargs: keyword arguments**

Keyword arguments among the following:

method: str If 'absolute' (default), amplitudes are absolute amplitudes in uV are returned. If 'relative', amplitudes are returned as ratios between waveform amplitudes and template amplitudes

peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

frames_before: int Frames before peak to compute amplitude

frames_after: int Frames after peak to compute amplitude

apply_filter: bool If True, recording is bandpass-filtered

freq_min: float High-pass frequency for optional filter (default 300 Hz)

freq_max: float Low-pass frequency for optional filter (default 6000 Hz)

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned

n_jobs: int Number of parallel jobs (default 1)

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

save_property_or_features: bool If true, it will save features in the sorting extractor

recompute_info: bool If True, waveforms are recomputed

max_spikes_per_unit: int The maximum number of spikes to extract per unit

seed: int Random seed for reproducibility

verbose: bool If True, will be verbose in metric computation

threshold sorting extractor

spiketoolkit.curation.**threshold_snrs**(*sorting*, *recording*, *threshold*, *threshold_sign*, *snr_mode='mad'*, *snr_noise_duration=10.0*, *max_spikes_per_unit_for_snr=1000*, *template_mode='median'*, *max_channel_peak='both'*, ***kwargs*)

Computes and thresholds the snrs in the sorted dataset with the given sign and value.

sorting: SortingExtractor The sorting result to be evaluated.

recording: RecordingExtractor The given recording extractor

threshold: int or float The threshold for the given metric.

threshold_sign: str If 'less', will threshold any metric less than the given threshold. If 'less_or_equal', will threshold any metric less than or equal to the given threshold. If 'greater', will threshold any metric greater than the given threshold. If 'greater_or_equal', will threshold any metric greater than or equal to the given threshold.

snr_mode: str Mode to compute noise SNR ('mad' | 'std' - default 'mad')

snr_noise_duration: float Number of seconds to compute noise level from (default 10.0)

max_spikes_per_unit_for_snr: int Maximum number of spikes to compute templates from (default 1000)

template_mode: str Use 'mean' or 'median' to compute templates

max_channel_peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

****kwargs: keyword arguments**

Keyword arguments among the following:

method: str If 'absolute' (default), amplitudes are absolute amplitudes in uV are returned. If 'relative', amplitudes are returned as ratios between waveform amplitudes and template amplitudes

peak: str If maximum channel has to be found among negative peaks ('neg'), positive ('pos') or both ('both' - default)

frames_before: int Frames before peak to compute amplitude

frames_after: int Frames after peak to compute amplitude

apply_filter: bool If True, recording is bandpass-filtered

freq_min: float High-pass frequency for optional filter (default 300 Hz)

freq_max: float Low-pass frequency for optional filter (default 6000 Hz)

grouping_property: str Property to group channels. E.g. if the recording extractor has the 'group' property and 'grouping_property' is 'group', then waveforms are computed group-wise.

ms_before: float Time period in ms to cut waveforms before the spike events

ms_after: float Time period in ms to cut waveforms after the spike events

dtype: dtype The numpy dtype of the waveforms

compute_property_from_recording: bool If True and 'grouping_property' is given, the property of each unit is assigned as the corresponding property of the recording extractor channel on which the average waveform is the largest

max_channels_per_waveforms: int or None Maximum channels per waveforms to return. If None, all channels are returned

n_jobs: int Number of parallel jobs (default 1)

memmap: bool If True, waveforms are saved as memmap object (recommended for long recordings with many channels)

save_property_or_features: bool If true, it will save features in the sorting extractor

recompute_info: bool If True, waveforms are recomputed

max_spikes_per_unit: int The maximum number of spikes to extract per unit

seed: int Random seed for reproducibility

verbose: bool If True, will be verbose in metric computation

threshold sorting extractor

class spiketoolkit.curation.**CurationSortingExtractor**(*parent_sorting*, *curation_steps=None*)

exclude_units(*unit_ids*)

This function deletes roots from the curation tree according to the given unit_ids

unit_ids: list or int The unit ids to be excluded

append_curation_step: bool Appends the curation step to the object keyword arguments

get_unit_ids()

This function returns a list of ids (ints) for each unit in the sorsted result.

unit_ids: array_like A list of the unit ids in the sorted result (ints).

get_unit_spike_train(*unit_id*, *start_frame=None*, *end_frame=None*)

This function extracts spike frames from the specified unit. It will return spike frames from within three ranges:

[start_frame, t_start+1, ..., end_frame-1] [start_frame, start_frame+1, ..., final_unit_spike_frame - 1] [0, 1, ..., end_frame-1] [0, 1, ..., final_unit_spike_frame - 1]

if both start_frame and end_frame are given, if only start_frame is given, if only end_frame is given, or if neither start_frame or end_frame are given, respectively. Spike frames are returned in the form of an array_like of spike frames. In this implementation, start_frame is inclusive and end_frame is exclusive conforming to numpy standards.

unit_id: int The id that specifies a unit in the recording

start_frame: int The frame above which a spike frame is returned (inclusive)

end_frame: int The frame below which a spike frame is returned (exclusive)

spike_train: numpy.ndarray An 1D array containing all the frames for each spike in the specified unit given the range of start and end frames

merge_units(*unit_ids*)
This function merges two roots from the curation tree according to the given unit_ids. It creates a new unit_id and root that has the merged roots as children.

unit_ids: list The unit ids to be merged

new_root_id: int The unit id of the new merged unit.

print_curation_tree(*unit_id*)
This function prints the current curation tree for the unit_id (roots are current unit ids).

unit_id: in The unit id whose curation history will be printed.

split_unit(*unit_id*, *indices*)
This function splits a root from the curation tree according to the given unit_id and indices. It creates two new unit_ids and roots that have the split root as a child. This function splits the spike train of the root by the given indices.

unit_id: int The unit id to be split

indices: list The indices of the unit spike train at which the spike train will be split.

new_root_ids: tuple A tuple of new unit ids after the split (integers).

9.3 Module spikeinterface.sorters

spikesorters.**available_sorters**()
Lists available sorters.

spikesorters.**get_default_params**(*sorter_name_or_class*)
Returns default parameters for the specified sorter.

sorter_name_or_class: str or SorterClass The sorter to retrieve default parameters from

default_params: dict Dictionary with default params for the specified sorter

spikesorters.**run_sorter**(*sorter_name_or_class*, *recording*, *output_folder=None*, *delete_output_folder=False*, *grouping_property=None*, *parallel=False*, *verbose=False*, *raise_error=True*, *n_jobs=-1*, *joblib_backend='loky'*, ***params*)
Generic function to run a sorter via function approach.

Two usages with name or class:

by name:

```
>>> sorting = run_sorter('tridesclous', recording)
```

by class:

```
>>> sorting = run_sorter(TridesclousSorter, recording)
```

sorter_name_or_class: str or SorterClass The sorter to retrieve default parameters from

recording: RecordingExtractor The recording extractor to be spike sorted

output_folder: str or Path Path to output folder

delete_output_folder: bool If True, output folder is deleted (default False)

grouping_property: str Splits spike sorting by 'grouping_property' (e.g. 'groups')

parallel: bool If True and spike sorting is by 'grouping_property', spike sorting jobs are launched in parallel

verbose: bool If True, output is verbose

raise_error: bool If True, an error is raised if spike sorting fails (default). If False, the process continues and the error is logged in the log file.

n_jobs: int Number of jobs when parallel=True (default=-1)

joblib_backend: str joblib backend when parallel=True (default='loky')

****params: keyword args** Spike sorter specific arguments (they can be retrieved with 'get_default_params(sorter_name_or_class)'

sortingextractor: SortingExtractor The spike sorted data

spikesorters.**run_sorters**(*sorter_list*, *recording_dict_or_list*, *working_folder*, *sorter_params={}*, *grouping_property=None*, *mode='raise'*, *engine=None*, *engine_kwargs={}*, *verbose=False*, *with_output=True*, *run_sorter_kwargs={}*)

Run several sorters on several recordings.

sorter_list: list of str List of sorter names to run.

recording_dict_or_list: dict or list A dict of recordings. The key will be the name of the recording. If a list is given then the name will be recording_0, recording_1, ...

working_folder: str The working directory. This must not exist before calling this function.

sorter_params: dict of dict with sorter_name as key This allow to overwrite default params for sorter.

grouping_property: str or None The property of grouping given to sorters.

mode: 'raise' or 'overwrite' or 'keep'

The mode when the subfolder of recording/sorter already exists.

- 'raise' : raise error if subfolder exists
- 'overwrite' : force recompute
- 'keep' : do not compute again if f=subfolder exists and log is OK

engine: 'loop' or 'multiprocessing' or 'dask'

Which approach to use to run the multiple sorters.

- 'loop' : run sorters in a loop (serially)

- 'multiprocessing' : use the Python multiprocessing library to run in parallel
- 'dask' : use the Dask module to run in parallel

engine_kwargs: dict

This contains kwargs specific to the launcher engine:

- 'loop' : no kargs
- 'multiprocessing' : {'processes' : } number of processes
- 'dask' : {'client':} the dask client for submiting task

verbose: bool Controls sorter verbosity.

with_output: bool return the output.

run_sorter_kwargs: dict

This contains kwargs specific to run_sorter function: * 'raise_error' [bool]

- 'parallel' : bool
- 'n_jobs' : int
- 'joblib_backend' : 'loky' / 'multiprocessing' / 'threading'

results [dict] The output is nested dict[(rec_name, sorter_name)] of SortingExtractor.

Using multiprocessing through this function does not allow for subprocesses, so sorters that already use internally multiprocessing will fail.

9.4 Module spikeinterface.comparison

spikecomparison.**compare_two_sorters**(*sorting1*, *sorting2*, *sorting1_name=None*, *sorting2_name=None*, *delta_time=0.4*, *sampling_frequency=None*, *match_score=0.5*, *chance_score=0.1*, *n_jobs=-1*, *verbose=False*)

Compares two spike sorter outputs.

- Spike trains are matched based on their agreement scores
- Individual spikes are labelled as true positives (TP), false negatives (FN), false positives 1 (FP from spike train 1), false positives 2 (FP from spike train 2), misclassifications (CL)

It also allows to get confusion matrix and agreement fraction, false positive fraction and false negative fraction.

sorting1: SortingExtractor The first sorting for the comparison

sorting2: SortingExtractor The second sorting for the comparison

sorting1_name: str The name of sorter 1

sorting2_name: [str] The name of sorter 2

delta_time: float Number of ms to consider coincident spikes (default 0.4 ms)

sampling_frequency: float Optional sampling frequency in Hz when not included in sorting

match_score: float Minimum agreement score to match units (default 0.5)

chance_score: float Minimum agreement score to for a possible match (default 0.1)

n_jobs: int Number of cores to use in parallel. Uses all available if -1

verbose: bool If True, output is verbose

sorting_comparison: SortingComparison The SortingComparison object

spikecomparison.**compare_multiple_sorters**(*sorting_list*, *name_list=None*, *delta_time=0.4*, *match_score=0.5*, *chance_score=0.1*, *n_jobs=-1*, *spiketrain_mode='union'*, *sampling_frequency=None*, *verbose=False*)

Compares multiple spike sorter outputs.

- Pair-wise comparisons are made
- An agreement graph is built based on the agreement score

It allows to return a consensus-based sorting extractor with the *get_agreement_sorting()* method.

sorting_list: list List of sorting extractor objects to be compared

name_list: list List of spike sorter names. If not given, sorters are named as 'sorter0', 'sorter1', 'sorter2', etc.

delta_time: float Number of ms to consider coincident spikes (default 0.4 ms)

match_score: float Minimum agreement score to match units (default 0.5)

chance_score: float Minimum agreement score to for a possible match (default 0.1)

n_jobs: int Number of cores to use in parallel. Uses all availible if -1

spiketrain_mode: str

Mode to extract agreement spike trains:

- 'union': spike trains are the union between the spike trains of the best matching two sorters
- **'intersection': spike trains are the intersection between the spike trains of the** best matching two sorters

sampling_frequency: float Sampling frequency (used if information is not in the sorting extractors)

verbose: bool if True, output is verbose

multi_sorting_comparison: MultiSortingComparison MultiSortingComparison object with the multiple sorter comparison

spikecomparison.**compare_sorter_to_ground_truth**(*gt_sorting*, *tested_sorting*, *gt_name=None*, *tested_name=None*, *delta_time=0.4*, *sampling_frequency=None*, *match_score=0.5*, *chance_score=0.1*, *well_detected_score=0.8*, *redundant_score=0.2*, *overmerged_score=0.2*, *exhaustive_gt=True*, *match_mode='hungarian'*, *n_jobs=-1*, *compute_labels=False*, *compute_misclassifications=False*, *verbose=False*)

Compares a sorter to a ground truth.

- Spike trains are matched based on their agreement scores
- Individual spikes are labelled as true positives (TP), false negatives (FN), false positives 1 (FP), misclassifications (CL)

It also allows to compute_performance and confusion matrix.

gt_sorting: SortingExtractor The first sorting for the comparison

tested_sorting: SortingExtractor The second sorting for the comparison

gt_name: str The name of sorter 1

tested_name: [str] The name of sorter 2

delta_time: float Number of ms to consider coincident spikes (default 0.4 ms)

sampling_frequency: float Optional sampling frequency in Hz when not included in sorting

match_score: float Minimum agreement score to match units (default 0.5)

chance_score: float Minimum agreement score to for a possible match (default 0.1)

redundant_score: float Agreement score above which units are redundant (default 0.2)

overmerged_score: float Agreement score above which units can be overmerged (default 0.2)

well_detected_score: float Agreement score above which units are well detected (default 0.8)

exhaustive_gt: bool (default True) Tell if the ground true is "exhaustive" or not. In other world if the GT have all possible units. It allows more performance measurement. For instance, MEArec simulated dataset have exhaustive_gt=True

match_mode: 'hungarian', or 'best' What is match used for counting : 'hugarian' or 'best match'.

n_jobs: int Number of cores to use in parallel. Uses all available if -1

compute_labels: bool If True, labels are computed at instantiation (default False)

compute_misclassifications: bool If True, misclassifications are computed at instantiation (default False)

verbose: bool If True, output is verbose

sorting_comparison: SortingComparison The SortingComparison object

class spikecomparison.**GroundTruthComparison**(*gt_sorting*, *tested_sorting*, *gt_name=None*, *tested_name=None*, *delta_time=0.4*, *sampling_frequency=None*, *match_score=0.5*, *well_detected_score=0.8*, *redundant_score=0.2*, *overmerged_score=0.2*, *chance_score=0.1*, *exhaustive_gt=False*, *n_jobs=-1*, *match_mode='hungarian'*, *compute_labels=False*, *compute_misclassifications=False*, *verbose=False*)

Class to compare a sorter to ground truth (GT)

This class can:

- compute a "macth between gt_sorting and tested_sorting
- compte th score label (TP, FN, CL, FP) for each spike
- count by unit of GT the total of each (TP, FN, CL, FP) into a Dataframe GroundTruthComparison.count
- compute the confusion matrix .get_confusion_matrix()
- compute some performance metric with several strategy based on the count score by unit
- count well detected units

- count false positve detected units
- count redundant units
- count overmerged units
- summary all this

count_bad_units()
See get_bad_units

count_false_positive_units(*redundant_score=None*)
See get_false_positive_units().

count_overmerged_units(*overmerged_score=None*)
See get_overmerged_units().

count_redundant_units(*redundant_score=None*)
See get_redundant_units().

count_well_detected_units(*well_detected_score*)
Count how many well detected units. Kargs are the same as get_well_detected_units.

get_bad_units()
Return units list of "bad units".

"bad units" are defined as units in tested that are not in the best match list of GT units.

So it is the union of "false positive units" + "redundant units".

Need exhaustive_gt=True

get_confusion_matrix()
Computes the confusion matrix.

confusion_matrix: pandas.DataFrame The confusion matrix

get_false_positive_units(*redundant_score=None*)
Return units list of "false positive units" from tested_sorting.

"false positive units" ara defined as units in tested that are not matched at all in GT units.

Need exhaustive_gt=True

redundant_score: float (default 0.2) The agreement score below which tested units are counted as "false positive"" (and not "redundant").

get_overmerged_units(*overmerged_score=None*)
Return "overmerged units"

"overmerged units" are defined as units in tested that match more than one GT unit with an agreement score larger than overmerged_score.

overmerged_score: float (default 0.4) Tested units with 2 or more agrement scores above 'overmerged_score' are counted as "overmerged".

get_performance(*method='by_unit'*, *output='pandas'*)

Get performance rate with several method:

- 'raw_count' : just render the raw count table
- 'by_unit' : render perf as rate unit by unit of the GT
- 'pooled_with_average' : compute rate unit by unit and average

method: str 'by_unit', or 'pooled_with_average'

output: str 'pandas' or 'dict'

perf: pandas dataframe/series (or dict) dataframe/series (based on 'output') with performance entries

get_redundant_units(*redundant_score=None*)
Return "redundant units"

"redundant units" are defined as units in tested that match a GT units with a big agreement score but it is not the best match. In other world units in GT that detected twice or more.

redundant_score=None: float (default 0.2) The agreement score above which tested units are counted as "redundant" (and not "false positive").

get_well_detected_units(*well_detected_score=None*)
Return units list of "well detected units" from tested_sorting.

"well detected units" ara defined as units in tested that are well matched to GT units.

well_detected_score: float (default 0.8) The agreement score above which tested units are counted as "well detected".

print_performance(*method='pooled_with_average'*)
Print performance with the selected method

print_summary(*well_detected_score=None*, *redundant_score=None*, *overmerged_score=None*)

Print a global performance summary that depend on the context:

- exhaustive= True/False
- how many gt units (one or several)

This summary mix several performance metrics.

class spikecomparison.**SymmetricSortingComparison**(*sorting1*, *sorting2*, *sorting1_name=None*, *sorting2_name=None*, *delta_time=0.4*, *sampling_frequency=None*, *match_score=0.5*, *chance_score=0.1*, *n_jobs=-1*, *verbose=False*)
Class for symmetric comparison of two sorters when no assumption is done.

get_agreement_fraction(*unit1=None*, *unit2=None*)

get_best_unit_match1(*unit1*)

get_best_unit_match2(*unit2*)

get_mapped_sorting1()
Returns a MappedSortingExtractor for sorting 1.

The returned MappedSortingExtractor.get_unit_ids returns the unit_ids of sorting 1.

The returned MappedSortingExtractor.get_mapped_unit_ids returns the mapped unit_ids of sorting 2 to the units of sorting 1 (if units are not mapped they are labeled as -1).

The returned MappedSortingExtractor.get_unit_spikeTrains returns the the spike trains of sorting 2 mapped to the unit_ids of sorting 1.

get_mapped_sorting2()
Returns a MappedSortingExtractor for sorting 2.

The returned MappedSortingExtractor.get_unit_ids returns the unit_ids of sorting 2.

The returned MappedSortingExtractor.get_mapped_unit_ids returns the mapped unit_ids of sorting 1 to the units of sorting 2 (if units are not mapped they are labeled as -1).

The returned MappedSortingExtractor.get_unit_spikeTrains returns the the spike trains of sorting 1 mapped to the unit_ids of sorting 2.

get_matching_event_count(*unit1*, *unit2*)

get_matching_unit_list1(*unit1*)

get_matching_unit_list2(*unit2*)

class spikecomparison.**GroundTruthStudy**(*study_folder=None*)

aggregate_count_units(*well_detected_score=None*, *redundant_score=None*, *overmerged_score=None*)

aggregate_dataframes(*copy_into_folder=True*, ***karg_thresh*)

aggregate_performance_by_units()

aggregate_run_times()

concat_all_snr()

copy_sortings()

classmethod create(*study_folder*, *gt_dict*)

get_ground_truth(*rec_name=None*)

get_recording(*rec_name=None*)

get_sorting(*sort_name*, *rec_name=None*)

get_units_snr(*rec_name=None*, ***snr_kargs*)
Load or compute units SNR for a given recording.

run_comparisons(*exhaustive_gt=False*, ***kwargs*)

run_sorters(*sorter_list*, *sorter_params={}*, *mode='keep'*, *engine='loop'*, *engine_kwargs={}*, *verbose=False*, *run_sorter_kwargs={'parallel': False}*)

scan_folder()

9.5 Module spikeinterface.widgets

spikewidgets.**plot_timeseries**(*recording*, *channel_ids=None*, *trange=None*, *color_groups=False*, *color=None*, *figure=None*, *ax=None*)

Plots recording timeseries.

recording: RecordingExtractor The recordng extractor object

channel_ids: list The channel ids to display.

trange: list List with start time and end time

color_groups: bool If True groups are plotted with different colors

color: matplotlib color, default: None The color used to draw the traces.

figure: matplotlib figure The figure to be used. If not given a figure is created

ax: matplotlib axis The axis to be used. If not given an axis is created

W: TimeseriesWidget The output widget

spikewidgets.**plot_electrode_geometry**(*recording*, *color='C0'*, *label_color='r'*, *figure=None*, *ax=None*)

Plots electrode geometry.

recording: RecordingExtractor The recordng extractor object

color: matplotlib color The color of the electrodes

label_color: matplotlib color The color of the channel label when clicking

figure: matplotlib figure The figure to be used. If not given a figure is created

ax: matplotlib axis The axis to be used. If not given an axis is created

W: UnitWaveformsWidget The output widget

spikewidgets.**plot_spectrum**(*recording*, *channels=None*, *trange=None*, *freqrange=None*, *color_groups=False*, *color='steelblue'*, *nfft=256*, *figure=None*, *ax=None*)

Plots electrode geometry.

recording: RecordingExtractor The recordng extractor object

channels: list The channels to show

trange: list List with start time and end time

freqrange: list List with start frequency and end frequency

color_groups: bool If True groups are plotted with different colors

color: matplotlib color The color to be used

figure: matplotlib figure The figure to be used. If not given a figure is created

ax: matplotlib axis The axis to be used. If not given an axis is created

W: TimeseriesWidget The output widget

spikewidgets.**plot_spectrogram**(*recording*, *channel*, *trange=None*, *freqrange=None*, *cmap='viridis'*, *nfft=256*, *figure=None*, *ax=None*)

Plots electrode geometry.

recording: RecordingExtractor The recordng extractor object

channel: int The channel to plot spectrogram of

trange: list List with start time and end time

freqrange: list List with start frequency and end frequency

cmap: matplotlib colorma The colormap to be used

figure: matplotlib figure The figure to be used. If not given a figure is created

ax: matplotlib axis The axis to be used. If not given an axis is created

W: TimeseriesWidget The output widget

spikewidgets.**plot_activity_map**(*recording*, *channel_ids=None*, *trange=None*, *activity='rate'*, *log=False*, *cmap='viridis'*, *background='on'*, *label_color='r'*, *transpose=False*, *frame=False*, *colorbar=False*, *colorbar_bbox=None*, *colorbar_orientation='vertical'*, *colorbar_width=0.02*, *ax=None*, *figure=None*, ***activity_kwargs*)

Plots spike rate (estimated using simple threshold detector) as 2D activity map.

recording: RecordingExtractor The recordng extractor object

channel_ids: list The channel ids to display

trange: list List with start time and end time

activity: str 'rate' or 'amplitude'. If 'rate' the channel spike rate is used. If 'amplitude' the spike amplitude is used

log: bool If True, log scale is used

cmap: matplotlib colormap The colormap to be used (default 'viridis')

background: bool If True, a background is added in between electrodes

transpose: bool, optional, default: False Swap x and y channel coordinates if True

frame: bool, optional, default: False Draw a frame around the array if True

colorbar: bool If True, a colorbar is displayed

colorbar_bbox: bbox Bbox (x,y,w,h) in figure coordinates to plot colorbar

colorbar_orientation: str 'vertical' or 'horizontal'

colorbar_width: float Width of colorbar in figure coordinates (default 0.02)

figure: matplotlib figure The figure to be used. If not given a figure is created

ax: matplotlib axis The axis to be used. If not given an axis is created

activity_kwargs: keyword arguments for st.postprocessing.compute_channel_spiking_activity()

W: ActivityMapWidget The output widget

spikewidgets.**plot_rasters**(*sorting*, *sampling_frequency=None*, *unit_ids=None*, *trange=None*, *color='k'*, *figure=None*, *ax=None*)

Plots spike train rasters.

sorting: SortingExtractor The sorting extractor object

sampling_frequency: float The sampling frequency (if not in the sorting extractor)

unit_ids: list List of unit ids

trange: list List with start time and end time

color: matplotlib color The color to be used

figure: matplotlib figure The figure to be used. If not given a figure is created

ax: matplotlib axis The axis to be used. If not given an axis is created

W: RasterWidget The output widget

spikewidgets.**plot_autocorrelograms**(*sorting*, *sampling_frequency=None*, *unit_ids=None*, *bin_size=2*, *window=50*, *figure=None*, *ax=None*, *axes=None*)

Plots spike train auto-correlograms.

sorting: SortingExtractor The sorting extractor object

sampling_frequency: float The sampling frequency (if not in the sorting extractor)

unit_ids: list List of unit ids

bin_size: float Bin size in s

window: float Window size in s

figure: matplotlib figure The figure to be used. If not given a figure is created

ax: matplotlib axis The axis to be used. If not given an axis is created

axes: list of matplotlib axes The axes to be used for the individual plots. If not given the required axes are created. If provided, the ax and figure parameters are ignored

W: AutoCorrelogramsWidget The output widget

spikewidgets.**plot_crosscorrelograms**(*sorting*, *sampling_frequency=None*, *unit_ids=None*, *bin_size=1*, *window=10*, *figure=None*, *ax=None*, *axes=None*)

Plots spike train cross-correlograms.

sorting: SortingExtractor The sorting extractor object

sampling_frequency: float The sampling frequency (if not in the sorting extractor)

unit_ids: list List of unit ids

bin_size: float Bin size in s

window: float Window size in s

figure: matplotlib figure The figure to be used. If not given a figure is created

ax: matplotlib axis The axis to be used. If not given an axis is created

axes: list of matplotlib axes The axes to be used for the individual plots. If not given the required axes are created. If provided, the ax and figure parameters are ignored

W: CrossCorrelogramsWidget The output widget

spikewidgets.**plot_isi_distribution**(*sorting*, *sampling_frequency=None*, *unit_ids=None*, *bins=10*, *window=1*, *figure=None*, *ax=None*, *axes=None*)

Plots spike train ISI distribution.

sorting: SortingExtractor The sorting extractor object

sampling_frequency: float The sampling frequency (if not in the sorting extractor)

unit_ids: list List of unit ids

bins: int Number of bins

window: float Window size in s

figure: matplotlib figure The figure to be used. If not given a figure is created

ax: matplotlib axis The axis to be used. If not given an axis is created

axes: list of matplotlib axes The axes to be used for the individual plots. If not given the required axes are created. If provided, the ax and figure parameters are ignored

W: ISIDistributionWidget The output widget

spikewidgets.**plot_unit_waveforms**(*recording*, *sorting*, *channel_ids=None*, *unit_ids=None*, *channel_locs=True*, *radius=None*, *max_channels=None*, *plot_templates=True*, *show_all_channels=True*, *color='k'*, *lw=2*, *axis_equal=False*, *plot_channels=False*, *set_title=True*, *figure=None*, *ax=None*, *axes=None*, ***waveforms_kwargs*)

Plots unit waveforms.

recording: RecordingExtractor The recording extractor object

sorting: SortingExtractor The sorting extractor object

channel_ids: list The channel ids to display

unit_ids: list List of unit ids.

max_channels: int Maximum number of largest channels to plot waveform

channel_locs: bool If True, channel locations are used to display the waveforms. If False, waveforms are displayed in vertical order (default)

plot_templates: bool If True, templates are plotted over the waveforms

radius: float If not None, all channels within a circle around the peak waveform will be displayed Ignores max_spikes_per_unit

set_title: bool Create a plot title with the unit number if True.

plot_channels: bool Plot channel locations below traces, only used if channel_locs is True

axis_equal: bool Equal aspext ratio for x and y axis, to visualise the array geometry to scale

lw: float Line width for the traces.

color: matplotlib color or list of colors Color(s) of traces.

show_all_channels: bool Show the whole probe if True, or only selected channels if False

figure: matplotlib figure The figure to be used. If not given a figure is created

ax: matplotlib axis The axis to be used. If not given an axis is created

axes: list of matplotlib axes The axes to be used for the individual plots. If not given the required axes are created. If provided, the ax and figure parameters are ignored

waveforms_kwargs: keyword arguments for st.postprocessing.get_unit_waveforms()

W: UnitWaveformsWidget The output widget

spikewidgets.**plot_unit_templates**(*recording*, *sorting*, *channel_ids=None*, *unit_ids=None*, *max_channels=None*, *channel_locs=True*, *radius=None*, *show_all_channels=True*, *color='k'*, *lw=2*, *axis_equal=False*, *plot_channels=False*, *set_title=True*, *figure=None*, *ax=None*, *axes=None*, ***template_kwargs*)

Plots unit waveforms.

recording: RecordingExtractor The recording extractor object

sorting: SortingExtractor The sorting extractor object

channel_ids: list The channel ids to display

unit_ids: list List of unit ids.

max_channels: int Maximum number of largest channels to plot waveform

channel_locs: bool If True, channel locations are used to display the waveforms. If False, waveforms are displayed in vertical order. (default)

radius: float If not None, all channels within a circle around the peak waveform will be displayed. Ignores max_spikes_per_unit

set_title: bool Create a plot title with the unit number if True

plot_channels: bool Plot channel locations below traces, only used if channel_locs is True

axis_equal: bool Equal aspext ratio for x and y axis, to visualise the array geometry to scale

lw: float Line width for the traces.

color: matplotlib color or list of colors Color(s) of traces.

show_all_channels: bool Show the whole probe if True, or only selected channels if False

figure: matplotlib figure The figure to be used. If not given a figure is created

ax: matplotlib axis The axis to be used. If not given an axis is created

axes: list of matplotlib axes The axes to be used for the individual plots. If not given the required axes are created. If provided, the ax and figure parameters are ignored

template_kwargs: keyword arguments for st.postprocessing.get_unit_templates()

W: UnitWaveformsWidget The output widget

spikewidgets.**plot_unit_template_maps**(*recording*, *sorting*, *channel_ids=None*, *unit_ids=None*, *peak='neg'*, *log=False*, *ncols=10*, *background='on'*, *cmap='viridis'*, *label_color='r'*, *figure=None*, *ax=None*, *axes=None*, ***templates_kwargs*)

Plots sorting comparison confusion matrix.

recording: RecordingExtractor The recordng extractor object

sorting: SortingExtractor The sorting extractor object

channel_ids: list The channel ids to display

unit_ids: list List of unit ids.

peak: str 'neg', 'pos' or 'both'

log: bool If True, log scale is used

ncols: int Number of columns if multiple units are displayed

background: str 'on' or 'off'

cmap: matplotlib colormap The colormap to be used (default 'viridis')

label_color: matplotlib color Color to display channel name upon click

figure: matplotlib figure The figure to be used. If not given a figure is created

ax: matplotlib axis The axis to be used. If not given an axis is created

axes: list of matplotlib axes The axes to be used for the individual plots. If not given the required axes are created. If provided, the ax and figure parameters are ignored

templates_kwargs: keyword arguments for st.postprocessing.get_unit_templates()

W: ActivityMapWidget The output widget

spikewidgets.**plot_amplitudes_distribution**(*recording*, *sorting*, *unit_ids=None*, *max_spikes_per_unit=100*, *figure=None*, *ax=None*, *axes=None*)

Plots waveform amplitudes distribution.

recording: RecordingExtractor The recording extractor object

sorting: SortingExtractor The sorting extractor object

unit_ids: list List of unit ids

max_spikes_per_unit: int Maximum number of spikes to display per unit

figure: matplotlib figure The figure to be used. If not given a figure is created

ax: matplotlib axis The axis to be used. If not given an axis is created

axes: list of matplotlib axes The axes to be used for the individual plots. If not given the required axes are created. If provided, the ax and figure parameters are ignored

W: AmplitudeDistributionWidget The output widget

spikewidgets.**plot_amplitudes_timeseries**(*recording*, *sorting*, *unit_ids=None*, *max_spikes_per_unit=100*, *figure=None*, *ax=None*, *axes=None*)

Plots waveform amplitudes timeseries.

recording: RecordingExtractor The recording extractor object

sorting: SortingExtractor The sorting extractor object

unit_ids: list List of unit ids

max_spikes_per_unit: int Maximum number of spikes to display per unit.

figure: matplotlib figure The figure to be used. If not given a figure is created

ax: matplotlib axis The axis to be used. If not given an axis is created

axes: list of matplotlib axes The axes to be used for the individual plots. If not given the required axes are created. If provided, the ax and figure parameters are ignored

W: AmplitudeTimeseriesWidget The output widget

spikewidgets.**plot_confusion_matrix**(*gt_comparison*, *count_text=True*, *unit_ticks=True*, *ax=None*, *figure=None*)

Plots sorting comparison confusion matrix.

gt_comparison: GroundTruthComparison The ground truth sorting comparison object

count_text: bool If True counts are displayed as text

unit_ticks: bool If True unit tick labels are displayed

figure: matplotlib figure The figure to be used. If not given a figure is created

ax: matplotlib axis The axis to be used. If not given an axis is created

W: ConfusionMatrixWidget The output widget

spikewidgets.**plot_agreement_matrix**(*sorting_comparison*, *ordered=True*, *count_text=True*, *unit_ticks=True*, *ax=None*, *figure=None*)

Plots sorting comparison confusion matrix.

sorting_comparison: GroundTruthComparison or SymmetricSortingComparison The sorting comparison object. Symetric or not.

ordered: bool Order units with best agreement scores. This enable to see agreement on a diagonal.

count_text: bool If True counts are displayed as text

unit_ticks: bool If True unit tick labels are displayed

figure: matplotlib figure The figure to be used. If not given a figure is created

ax: matplotlib axis The axis to be used. If not given an axis is created

W: ConfusionMatrixWidget The output widget

spikewidgets.**plot_sorting_performance**(*gt_sorting_comparison*, *property_name=None*, *metric='accuracy'*, *markersize=10*, *marker='.'*, *figure=None*, *ax=None*)

Plots sorting performance for each ground-truth unit.

gt_sorting_comparison: GroundTruthComparison The ground truth sorting comparison object

property_name: str The property of the sorting extractor to use as x-axis (e.g. snr). If None, no property is used.

metric: str The performance metric. 'accuracy' (default), 'precision', 'recall', 'miss rate', etc.

markersize: int The size of the marker

marker: str The matplotlib marker to use (default '.')

figure: matplotlib figure The figure to be used. If not given a figure is created

ax: matplotlib axis The axis to be used. If not given an axis is created

W: SortingPerformanceWidget The output widget

spikewidgets.**plot_multicomp_graph**(*multi_sorting_comparison*, *draw_labels=False*, *node_cmap='viridis'*, *edge_cmap='hot_r'*, *alpha_edges=0.7*, *colorbar=False*, *figure=None*, *ax=None*)

Plots multi sorting comparison graph.

multi_sorting_comparison: MultiSortingComparison The multi sorting comparison object

draw_labels: bool If True unit labels are shown

node_cmap: matplotlib colormap The colormap to be used for the nodes (default 'viridis')

edge_cmap: matplotlib colormap The colormap to be used for the edges (default 'hot')

alpha_edges: float Alpha value for edges

colorbar: bool If True a colorbar for the edges is plotted

figure: matplotlib figure The figure to be used. If not given a figure is created

ax: matplotlib axis The axis to be used. If not given an axis is created

W: MultiCompGraphWidget The output widget

spikewidgets.**plot_multicomp_agreement**(*multi_sorting_comparison*, *plot_type='pie'*, *cmap='YlOrRd'*, *figure=None*, *ax=None*)

Plots multi sorting comparison agreement as pie or bar plot.

multi_sorting_comparison: MultiSortingComparison The multi sorting comparison object

plot_type: str 'pie' or 'bar'

cmap: matplotlib colormap The colormap to be used for the nodes (default 'Reds')

figure: matplotlib figure The figure to be used. If not given a figure is created

ax: matplotlib axis The axis to be used. If not given an axis is created

W: MultiCompGraphWidget The output widget

spikewidgets.**plot_multicomp_agreement_by_sorter**(*multi_sorting_comparison*, *plot_type='pie'*, *cmap='YlOrRd'*, *figure=None*, *ax=None*, *axes=None*, *show_legend=True*)

Plots multi sorting comparison agreement as pie or bar plot.

multi_sorting_comparison: MultiSortingComparison The multi sorting comparison object

plot_type: str 'pie' or 'bar'

cmap: matplotlib colormap The colormap to be used for the nodes (default 'Reds')

figure: matplotlib figure The figure to be used. If not given a figure is created

ax: matplotlib axis A single axis used to create a matplotlib gridspec for the individual plots. If None, an axis will be created.

axes: list of matplotlib axes The axes to be used for the individual plots. If not given the required axes are created. If provided, the ax and figure parameters are ignored.

show_legend: bool Show the legend in the last axes (default True).

W: MultiCompGraphWidget The output widget

CHAPTER 10

Release notes

10.1 SpikeInterface 0.13.0 release notes

13th Mar 2021

- spikeextractors: https://github.com/SpikeInterface/spikeextractors/releases/tag/0.9.7
- spiketoolkit: https://github.com/SpikeInterface/spiketoolkit/releases/tag/0.7.6
- spikesorters: https://github.com/SpikeInterface/spikesorters/releases/tag/0.4.5
- spikecomparison: https://github.com/SpikeInterface/spikecomparison/releases/tag/0.3.3
- spikewidgets: https://github.com/SpikeInterface/spikewidgets/releases/tag/0.5.4

10.2 SpikeInterface 0.12.0 release notes

13th Mar 2021

- spikeextractors: https://github.com/SpikeInterface/spikeextractors/releases/tag/0.9.5
- spiketoolkit: https://github.com/SpikeInterface/spiketoolkit/releases/tag/0.7.3
- spikesorters: https://github.com/SpikeInterface/spikesorters/releases/tag/0.4.4
- spikecomparison: https://github.com/SpikeInterface/spikecomparison/releases/tag/0.3.2
- spikewidgets: https://github.com/SpikeInterface/spikewidgets/releases/tag/0.5.2

10.3 SpikeInterface 0.11.0 release notes

10th Dec 2020

- spikeextractors: https://github.com/SpikeInterface/spikeextractors/releases/tag/0.9.3

- spiketoolkit: https://github.com/SpikeInterface/spiketoolkit/releases/tag/0.7.2
- spikesorters: https://github.com/SpikeInterface/spikesorters/releases/tag/0.4.3
- spikecomparison: https://github.com/SpikeInterface/spikecomparison/releases/tag/0.3.1
- spikewidgets: https://github.com/SpikeInterface/spikewidgets/releases/tag/0.5.1

10.4 SpikeInterface 0.10.0 release notes

28th Aug 2020

- **spikeextractors: https://github.com/SpikeInterface/spikeextractors/releases/tag/0.9.0**
  - Support for pynwb >= 1.3.3
  - Improved NWB sorting extractor
  - Added NeuroscopeRecordingExtractor
  - Added HDsort sorting extractor
  - Added WaveClusSortingExtractor
  - Various bug fixes
- **spiketoolkit: https://github.com/SpikeInterface/spiketoolkit/releases/tag/0.7.0**
  - Various fixes and improvements to the export_to_phy function
  - Improved performance of compute_amplitudes
  - Fix indexing bug in validation and curation when a subset of units is used
- **spikesorters: https://github.com/SpikeInterface/spikesorters/releases/tag/0.4.2**
  - Improved logging and verbose system
  - Fixed bugs for parallel processing by having supported backends for eash sorter
  - Updates to WaveClus, HDSort
  - Fix param bug in Kilosort
  - Check installed sorters with dynamic is_installed() function
- **spikecomparison: https://github.com/SpikeInterface/spikecomparison/releases/tag/0.3.0**
  - Fixed bug in GT study due to changes in spiketoolkit
  - Updates to reflect changes in spikesorters
- **spikewidgets: https://github.com/SpikeInterface/spikewidgets/releases/tag/0.5.0**
  - Improved map widgets with possibility to add bounding box
  - Made interactive plots to show channel labels upom clicking
  - Imprived handling of multiple axes in BaseMultiWidget

10.5 SpikeInterface 0.9.9 release notes

14th May 2020

- **spikeextractors:**
  - bug fixes
  - introduced dumping mechanism
  - introduced several new extractors
- **spiketoolkit:**
  - bug fixes
  - introduced dumping mechanism
  - improved postprocessing performance with memmap and parallel processing
  - introduced template feature extraction
  - refractoring of validation and curation modules
  - extended curation to all quality metrics
- **spikesorters:**
  - bug fixes
  - introduced dask engine for server processing
- **spikecomparison:**
  - refractor of multisortingcomparison
- **spikewidgets:**
  - bug fixes
  - improved several widgets
  - added plot_activity_maps, plot_unit_template_maps, plot_multicomp_agreement, and plot_multicomp_agreement_by_sorter widgets

10.6 SpikeInterface 0.9.1 release notes

7th October 2019

First release of SpikeInterface, a unified Python framework for spike sorting.

10.7 Version 0.13.0

- Final release of version 0.1X - bug fixes

10.8 Version 0.12.0

- Major update: API change for get_traces to enable return_scaled

10.9 Version 0.11.0

- Bug fixes and improvements on efficiency

10.10 Version 0.10.0

- Minor updates and bug fixes for biorXiv preprint

10.11 Version 0.9.9

- Major updates and bug fixes to all packages - pre-release

10.12 Version 0.9.1

- First SpikeInterface pre-release

CHAPTER 11

Contact Us

The following people have contributed code and/or ideas to the current version of SpikeInterface. The institutional affiliations are those at the time of the contribution, and may not be the current affiliation of a contributor.

- Alessio Paolo Buccino [1]
- Cole Hurwitz [2]
- Jeremy Magland [3]
- Matthias Hennig [2]
- Samuel Garcia [4]
- Josh Siegle [5]

For any inquiries, please contact Alessio Buccino (alessiop.buccino@gmail.com), Cole Hurwitz (cole.hurwitz@ed.ac.uk), or just write an issue!

1. Bio Engineering Laboratory, Dept. of Biosystems Science and Engineering, ETH Zurich, Switzerland.
2. The Institute for Adaptive and Neural Computation (ANC), University of Edinburgh, Edinburgh, Scotland.
3. Center for Computational Biology (CCB), Flatiron Institute, New York, United States.
4. Centre de Recherche en Neuroscience de Lyon (CRNL), Lyon, France.
5. Allen Institute for Brain Science, Washington, United States.

For more information, please have a look at:

- The eLife paper
- 1-hour video tutorial, recorded for the NWB User Days (Sep 2020)
- A collection of analysis notebook SpikeInterface Reports

Python Module Index

S

Index

D

E

F

G

L

M

N

P

R

S

T

W