Neuroinform (2015) 13:471–486
DOI 10.1007/s12021-015-9271-8

SOFTWARE ORIGINAL ARTICLE

Wyrm: A Brain-Computer Interface Toolbox in Python

Bastian Venthur[1] · Sven Dähne[1,2,3] · Johannes Höhne[1] · Hendrik Heller[1] · Benjamin Blankertz[1]

Published online: 24 May 2015

Abstract In the last years Python has gained more and more traction in the scientific community. Projects like NumPy, SciPy, and Matplotlib have created a strong foundation for scientific computing in Python and machine learning packages like scikit-learn or packages for data analysis like Pandas are building on top of it. In this paper we present Wyrm (https://github.com/bbci/wyrm), an open source BCI toolbox in Python. Wyrm is applicable to a broad range of neuroscientific problems. It can be used as a toolbox for analysis and visualization of neurophysiological data and in real-time settings, like an online BCI application. In order to prevent software defects, Wyrm makes extensive use of unit testing. We will explain the key aspects of Wyrm's software architecture and design decisions for its data structure, and demonstrate and validate the use of our toolbox by presenting our approach to the classification tasks of two different data sets from the BCI Competition III. Furthermore, we will give a brief analysis of the data sets using our toolbox, and demonstrate how we implemented an online experiment using Wyrm. With Wyrm we add the final piece to our ongoing effort to provide a complete, free and open source BCI system in Python.

✉ Bastian Venthur
bastian.venthur@tu-berlin.de

[1] Department of Neurotechnology, Technische Universität Berlin, Sekr. MAR 4-3 Marchstraße 23, 10587 Berlin, Germany

[2] Department of Machine Learning, Technische Universität, Berlin, Germany

[3] Bernstein Center for Computational Neuroscience, Berlin, Germany

Keywords Brain-computer interface · BCI · EEG · ECoG · Toolbox · Python · Machine learning · Signal processing

Introduction

Python is currently amongst the most popular programming languages (Louden et al. 2011; Bissyandé et al. 2013) and has become an important platform for scientific computing. Open source projects like NumPy, SciPy (Oliphant 2007; Jones et al. 2001), Matplotlib (Hunter 2007), IPython (Pérez and Granger 2007) have become the foundation of scientific computing in Python and other projects like Scikit-learn (Pedregosa et al. 2011) for machine learning or Pandas (McKinney 2012) for data analysis are building on top of them. Python is free- and open source software, and runs on most platforms, which makes it attractive for research institutions and substantially lowers the entry barrier for newcomers to the field.

Yet, in the brain-computer interface (BCI) community Matlab is still prevalent. Many toolboxes have been developed over the years to cover the various needs and research interests. One of the oldest toolboxes is BioSig (Schlögl and Brunner 2008) which is mainly for offline analysis of various biosignals, including EEG and ECoG data. BioSig is running in Matlab and Octave but experimental bindings for other programming languages exist. BioSig is free and open source software. The BBCI toolbox (http://bbci.de/toolbox) is also a Matlab toolbox which has matured with age. The BBCI toolbox is suitable for online experiments and offline data analysis and has been open sourced in 2012. It allows complex online processing chains, e.g., acquiring data from several data sources that operate with different sampling rates, to use different feature extraction methods and

classifiers simultaneously and to implement adaptive feature extraction and classifiers. FieldTrip (Oostenveld et al. 2011) is an open source Matlab toolbox for MEG and EEG analysis. It is relatively new but has already gained a lot of attention in the community. BCILAB (Kothe and Makeig 2013) is the latest open source Matlab toolbox for BCI research. It supports offline analysis and online experiments. Interesting toolboxes also exist outside the Matlab community: BCI2000 (Schalk et al. 2004) is a general purpose BCI system. It is written in C++ and its use is free for non-profit research and educational purposes. The open source software OpenViBE (Renard et al. 2010) has a special approach, as it allows for a visual programming of BCI paradigms. It also has Python and Matlab bindings and is licensed under the terms of the Affero General Public License (AGPL).

In Python we have BCPy2000 (Schreiner et al. 2008), which allows for writing BCI2000 modules in Python instead of C++, leveraging the infrastructure of BCI2000 without forcing the user to program in C++. A relatively new toolbox is pySPACE (Krell et al. 2013), a signal processing and classification environment in Python. pySPACE has implemented many signal processing algorithms and allows for conducting experiments without programming by providing configuration files for each experiment. Due to its modular design it allows for implementing own algorithms as well. pySPACE is suitable for offline analysis and online classification and licensed under the terms of the GPL. There is also OpenBCI (http://openbci.pl), a BCI system in Python. This project provides drivers for a few EEG amplifiers, tools for displaying and storing EEG signals and tools for creating bindings for 3rd party software for performing experiments. The project accumulated quite a lot of code but is unfortunately seemingly discontinued as the last commit in the repository was in 2011. MNE-Python (Gramfort et al. 2013) allows for offline analysis of MEG and EEG data and is available under the terms of the BSD license. SCoT is a special purpose toolbox for EEG source connectivity in Python licensed under the terms of the MIT license.

For more in depth information on related BCI software, see Brunner et al. (2013).

In this paper we introduce our toolbox Wyrm. Together with Mushu (Venthur and Blankertz 2012) for signal acquisition and Pyff (Venthur et al. 2010) for feedback- and stimulus presentation, Wyrm is the final step in our ongoing effort to create a complete, open source BCI system in Python.

The rest of the paper is divided into the following parts: in the next section we will give a slightly technical overview of the toolbox, including the design of the main data structure, an overview of the functions, and some means of quality assurance we have taken. In the Sections "Classification of Motor Imagery in ECoG Recordings" and "ERP Component Classification in EEG Recordings" we will demonstrate how we perform the classification task on two different data sets from the BCI Competition III (Blankertz et al. 2006). One data set is about classification of imagined pinky and tongue movement using ECoG recordings, the other is about classifying event-related potentials (ERPs) from a matrix speller using EEG recordings. We also will show some brief analysis of the data and present the results of the classification. In Section "Performing Online- and Simulated Online Experiments" we will demonstrate how to conduct an online experiment using Wyrm and in Section "Performance" we will analyze Wyrm's realtime capabilities and performance limitations. Finally, we will discuss the results and conclude the paper.

Toolbox Architecture

In this section we will give an introduction into the technical details of the toolbox. We will explain the main data structure that is used throughout the toolbox, show an overview of the toolbox functions, discuss performance concerns, explain how we utilize unit testing as a mean of quality assurance, and how the extensive documentation is created.

Data Structures

In order to work efficiently with the toolbox, it is necessary to understand the toolbox' main data structure, dubbed `Data`. It is used in almost all functions of the toolbox and fortunately it is not very difficult. Before we can begin, we have to explain the terminology that is used in NumPy and thus throughout our toolbox for describing n-dimensional arrays. A NumPy array is a table of elements of the same type, indexed by positive integers. The *dimensions* of an array are sometimes called *axes*. For example: an array with n rows and m columns has two dimensions (axes), the first dimension having the *length* n and the second the length m. The *shape* of an array is a tuple indicating the length (or size) of each dimension. The length of the shape tuple is therefore the number of dimensions of the array. Let's assume we have an EEG recording with 1000 data points and 32 channels. We could store this data in a `[time, channel]` array. This array would have two dimensions and the shape (1000, 32). The time axis would have the length of 1000 and the channel axis the length of 32.

For the design of the data structure it is essential to take into account that the functions would deal with many kinds of data, such as continuous multi-channel EEG recordings, epoched data of different kinds, spectrograms, spectra, feature vectors, and many more. What all those types of

data have in common is that they are representable as n-dimensional data. What separates them, from a data structure point of view, is merely the number of dimensions and the different names (and meanings) of their axes. We decided to create a simple data structure which has an n-dimensional array to store the data at its core, and a small set of meta information to describe the data sufficiently. Those extra attributes are: *names*, *axes*, and *units*. The `names` attribute is used to store the quantities or names for each dimension in the data. For example: a multi-channel spectrogram has the dimensions: (time, frequency, channel), consequently would the `names` attribute be an array of three strings: `['time', 'frequency', 'channel']`. The order of the elements in the `names` attribute corresponds to the order of the dimensions in the `Data` object: the first element belongs to the first dimension of the data, and so on. The `axes` attribute describes the rows and columns of the data, like headers describe the rows and columns of a table. It is an array of arrays. The length of the `axes` array is equal to the number of dimensions of the data, the lengths of the arrays inside correspond to the shape of the data. For the spectrogram, the first array would contain the times, the second the frequencies and the third the channel names of the data. The last attribute, `units` contains the (preferably) physical units of the data in `axes`. For the spectrogram that array would be: `['ms', 'Hz', '#']` (Since the channel names have no physical unit we use the hash (#) sign to indicate that the corresponding axis contains labels).

These three attributes are mandatory. It is tempting to add more meta information to describe the data even better, but more metadata adds more complexity to the toolbox functions in order to maintain consistency. So there is a trade-off between completeness of information and complexity of the code. Since complex (or more) code is harder to understand, harder to maintain and tends to have more bugs (Lipow 1982), we decided for a small set of obligatory metadata to describe the data sufficiently and make the toolbox pleasant to use, without the claim to provide a data structure that is completely self-explaining on its own.

Keeping the data structure simple and easy to understand was an important design decision. The rationale behind this decision was that is must be clear what is stored in the data structure, and where, to encourage scientists to not only look at the data in different ways, but also manipulate at it at will without the data structure getting in the way. It was also clear that specific experiments have specific requirements for the information being stored, since we cannot anticipate all future use cases of the toolbox, it was important for us to allow the data structure to be easily extended, so users can add more information to the data structure if needed. Consequently, we designed all toolbox functions to ignore unknown attributes and more importantly, to never remove any additional information from `Data` objects.

To summarize, Wyrm's main data structure (visualized in Fig. 1), the `Data` class, has the following attributes: `.data`, which contains arbitrary, n-dimensional data, `.axes` which contains the headers for the columns of the data, `.names` which contains the of names the axes of `.data`, and `.units` which contains the units for the values in `.axes`. The `Data` class has some more functionality, for example built-in consistency checking to test whether the lengths of the attributes are compatible. This data structure is intentionally generic enough to contain many kinds of data, even data the authors of this paper did not anticipate during the design. Whenever additional information is needed, it can be easily added to the `Data` class by means of subclassing or by simply adding it to existing `Data` objects, thanks to the dynamic nature of Python.

Wyrm also implements two other data structures: a ring buffer and a block buffer. Those data structures are useful in online experiments and are demonstrated in Section "Performing Online- and Simulated Online Experiments".

Toolbox Functions

Our toolbox implements dozens of functions, covering a broad range of aspects for offline analysis and online applications. The list of algorithms includes: channel selection,

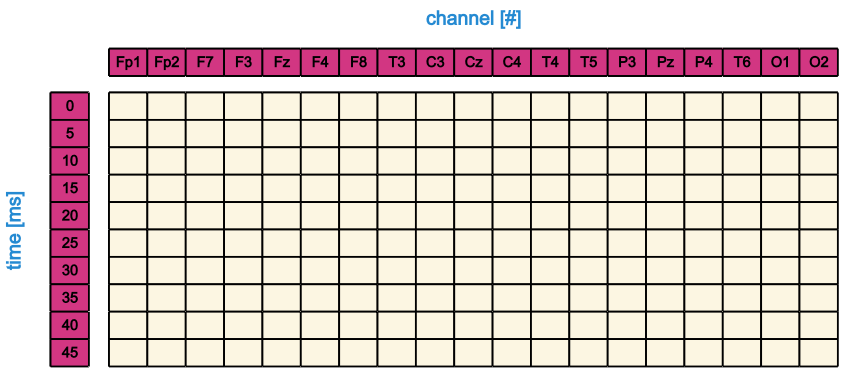

Fig. 1 Visualization of the `Data` object and its attributes. In this example the data is two dimensional (*yellow block*). The `axes` (*magenta*) describe the rows and columns of the data and the `names` and `units` (*blue*) are the headings of the table

IIR filters, sub-sampling, spectrograms, spectra, baseline removal for signal processing, Common Spatial Patterns (CSP) (Ramoser et al. 2000), Source Power Co-modulation (SPoC) (Dähne et al. 2014), classwise average, jumping means, signed r^2-values for feature extraction, Linear Discriminant Analysis (LDA) with and without shrinkage for machine learning (Blankertz et al. 2011), various plotting functions and many more. Wyrm's `io` module also provides a few input/output functions for foreign formats. Currently supported file formats are EEG files from Brain Products and from the Mushu signal acquisition, reading data from amplifiers supported by Mushu, and two functions specifically written to load the BCI competition data sets used in Sections "Classification of Motor Imagery in ECoG Recordings", "ERP Component Classification in EEG Recordings", and "Performing Online- and Simulated Online Experiments". For a complete overview, please refer to Wyrm's documentation (http://bbci.github.io/wyrm/).

It is worth mentioning that with scikit-learn (Pedregosa et al. 2011) you have a wide range of machine learning algorithms readily at your disposal. This list includes: cross validation, Support Vector Machines (SVM), k-Nearest Neighbours (KNN), Independent- and Principal Component Analysis (ICA, PCA), Gaussian Mixture Models (GMM), Kernel Regression, and many more. Our data format (Section "Data Structures") is compatible with scikit-learn and one can mostly apply the algorithms without any data conversion step at all.

Almost all functions operate on `Data` objects introduced in Section "Data Structures" and are responsible for keeping the data *and* the metadata consistent. While a few functions like `square`, `variance`, or `logarithm` are just convenient wrappers around the respective NumPy equivalents that accept `Data` object instead of NumPy arrays, the vast majority of functions implement a lot more functionality. For example the function `select_channels` requires a `Data` object and a list of strings as parameters. The strings can be channel names or regular expressions that are matched against the channel names in the `Data` object's metadata. `select_channels` will not only return a copy of the `Data` object with all channels removed that where not part of the list, it will also make sure the metadata that contains the channel names for the returned `Data` object is correctly updated. This approach is less error prone and much easier to read, than doing the equivalent operations on the data and metadata separately.

To ease the understanding of the processing functions, special attention was paid to keep syntax and semantics of the functions consistent. We also made sure that the user can rely on a set of features shared by all functions of the toolbox. For example: functions never modify their input arguments. They create a deep copy of them and return a possibly modified version of that copy if necessary. This encourages a functional style of programming which, in our opinion, is well suited when delving into the data:

```
>>> from wyrm import processing as proc
>>> # as a convention we import the processing module
>>> # as proc
>>>
>>> dat2 = proc.some_function(dat)
>>> # dat is still the same and not modified
>>>
>>> dat = proc.some_other_function(dat)
>>> # dat was replaced by the result
```

A function never touches attributes of a `Data` object which are unrelated to the functionality of that function. In particular, a function never removes custom or unknown attributes:

```
>>> # we create a new attribute on the fly:
>>> dat.some_obscure_attribute = "foo"
>>> dat2 = proc.some_function(dat)
>>> # the attribute is still present in the
>>> # result of some_function
>>> dat2.some_obscure_attribute
"foo"
```

If a function operates on a specific axis of a `Data` object (Section "Data Structures"), it adheres by default to our convention, but gives the option to change the index of the axis to operate on by means of Python's default arguments. Those default arguments are clearly named as `timeaxis`, or `classaxis`, etc.:

```
>>> # create a copy of dat -> dat2
>>> dat2 = dat.copy()
>>> # swap the first two axes of dat2
>>> dat2 = proc.swap_axes(dat2, 0, 1)
>>> # channel axis is by convention the last one
>>> dat = proc.remove_channels(dat,
... ['Cz', 'O1', 'O2'])
>>> # if not, we have to tell `remove_channels` on
>>> # which axis the channels are
>>> dat2 = proc.remove_channels(dat,
... ['Cz', 'O1', 'O2'], chanaxis=0)
>>> dat == dat2
True
```

In Sections "Classification of Motor Imagery in ECoG Recordings", "ERP Component Classification in EEG Recordings", and "Performing Online- and Simulated Online Experiments", you will find some realistic examples of the usage of our toolbox and its functions.

Speed

We realize that speed is an important factor in scientific computation, especially for online experiments, were one iteration of the main loop must not take longer than the duration of the samples being processed in that iteration. One drawback of dynamic languages like Python or Ruby is the slow execution speed compared to compiled languages like

C or Java. This issue is particularly important in scientific computing, where non-trivial computations in Python can easily be in the order of two or more magnitudes slower than the equivalent implementations in C. The main reason for the slow execution speed is the dynamic type system: since variables in Python have no fixed type and can change at any time during the execution of the program, the Python interpreter has to check the types of the involved variables for compatibility before every single operation.

NumPy mitigates this problem by providing statically typed arrays and fast operations on them. When used properly, this allows for almost C-like execution speed in Python programs. In Wyrm all data structures use NumPy arrays internally and Wyrm's toolbox functions use NumPy or SciPy operations on those data structures. We also carefully profiled our functions in order to find and eliminate bottlenecks in execution speed. Wyrm is thus very fast and suitable even for online experiments, as we will demonstrate in the Sections "Performing Online- and Simulated Online Experiments" and "Performance".

Unit Tests and Continuous Integration

Since the correctness of its functions is crucial for a toolbox, we used unit testing to ensure all functions work as intended. The concept of unit testing is to write tests for small, individual units of code (usually single functions). These tests ensure that the tested function meets its design and is fit for use. Typically, a test will simply call the tested function with defined arguments and compare the returned result with the expected result. If both are equal the test passes, if not it fails. Well written tests are independent of each other and treat the tested method as a black box by not making any assumptions about how the function works, but only comparing the expected result with the actual one. Those tests should be organized in a way that makes it easy to run all tests at once with little effort (usually a single command). This encourages developers to run tests often. When done properly, unit tests facilitate refactoring of the code base (i.e. restructuring the code without changing its functionality), speed up development time significantly, and reduce the number of bugs.

In our toolbox *each* method is tested respectively by a handful of test cases which ensure that the functions calculate the correct results, throw the expected errors if necessary, do not modify the input arguments, work with non-conventional ordering of axis, etc. The total amount of code for all tests is roughly 2-3 times bigger than the amount code for the toolbox functions. This is not unusual for software projects.

To automate the testing even further, we use a continuous integration (CI) service in conjunction with Wyrm's github repository. Whenever a new version is pushed to github, the CI will run the unit tests with three different Python versions (2.7, 3.3, and 3.4) to verify that all tests still pass. If and only if the unit tests pass with all three Python versions, the revision counts as passing, otherwise the developers will get a notification via mail. The whole CI process is fully automated and requires no interaction.

Documentation

A software toolbox would be hard to use without proper documentation. We provide documentation that consists of readable prose and extensive API documentation (http://bbci.github.io/wyrm/). The first part consists of a high level introduction to the toolbox, explaining the conventions and terminology being used, as well as tutorials how to write your own toolbox functions. The second part, the API documentation, is generated from special comments in the source code of the tool box, so called docstrings (Goodger and van Rossum 2001). External documentation of software tends to get outdated as the software evolves. Therefore, having documentation directly in the source code of the respective module, class, or method is an important mean to keep the documentation and the actual behaviour of the code consistent. Each method of the toolbox is extensively documented. Usually a method has a short summary, a detailed description of the algorithm, a list of expected inputs, return values and exceptions, as well as cross references to related functions in- or outside the toolbox and example code to demonstrate how to use the method. All this information is written within the docstring of the method (i.e. in the actual source code) and HTML or PDF documentation can be generated for the whole toolbox with a single command. The docstrings are also used by Python's interactive help system.

Python 2 versus Python 3

By the end of 2008 Python 3 was released. Python 3 was intentionally not backwards compatible with Python 2, in order to fix some longstanding design problems with Python 2. Since the porting of Python 2 software to Python 3 is not trivial for bigger projects, the adoption of Python 3 gained momentum only slowly. Although Python 2.7 is the last version of the 2.x series, it still receives backwards compatible bug fixes and enhancements. This is certainly a responsible decision by the Python developers but probably one of the reasons for the slow adoption of Python 3. As of today, most of the important packages have been ported to Python 3, but there is still a bit of a divide between the Python 2 and Python 3 packages.

We decided to support both Python versions. Wyrm is mainly developed under Python 2.7, but written in a *forward compatible* way to support Python 3 as well. Our unit

tests ensure that the functions provide the expected results in Python 2.7, Python 3.3, and Python 3.4.

Classification of Motor Imagery in ECoG Recordings

To demonstrate the usage of our toolbox we describe the analysis and classification of two data sets from the BCI Competition III (Blankertz 2005) using our toolbox. The scripts we will show are included in the `examples` directory of the Wyrm toolbox and the data sets are freely available on the BCI Competition III homepage (http://www.bbci.de/competition/iii). The reader can reproduce our results by using the scripts and the data sets.

The following code examples in this and the next section follow our convention to import Wyrm's `processing` module as `proc`:

```
from wyrm import processing as proc
```

The first data set uses Electrocorticography (ECoG) recordings, provided by the Eberhard-Karls-Universität Tübingen, and the Max-Planck-Institute for Biological Cybernetics, Täbingen, Germany, cf. (Lal et al. 2005). The time series were recorded using a 8x8 ECoG platinum gird which was placed on the contralateral, right motor cortex. The grid covered the motor cortex completely, but also surrounding cortex areas due to is size of approximately 8x8cm. All data was recorded with a sampling frequency of 1kHz and the data was stored as μV values. During the experiment the subject had to perform imagined movements of either the left small finger or the tongue. Each trial consisted of either an imagined finger- or tongue movement and was recorded for a duration of 3 seconds. The recordings in the data set start at 0.5 seconds after the visual cue had ended in order to avoid visual evoked potentials (Lal et al. 2005). It is worth noting that the training- and test data were recorded on the same subject but with roughly one week between both recordings.

The data set consists of 278 trials of training data and 100 trials of test data. During the BCI Competition only the labels (finger or tongue movement) for the training data were available. The task for the competition was to use the training data and its labels to predict the 100 labels of the test data. Since the competition is over, we also had the true labels for the test data, so we could calculate and compare the accuracy of our results.

As part of the signal processing chain in this example, we employ a spatial filtering technique called Common Spatial Patterns (CSP) (Ramoser et al. 2000; Blankertz et al. 2008). CSP spatial filters are applied to band-pass filtered data. The outputs of the spatial filters (sometimes also referred to as CSP components) are then used in subsequent processing steps. The main advantage of the CSP algorithm is that the filter coefficients are optimized to maximize the difference in variance between two classes. Trial-wise variance of band-passed filtered signals approximates the spectral power in the pass-band and thus improves the detectability of event-related (de-)synchronization (ERD/ERS), which in turn represents the basis for motor imagery BCI applications.

For classification we will use Linear Discriminant Analysis (LDA) (Blankertz et al. 2011). LDA is a simple and robust linear classification method which is frequently applied for BCI data.

After initial conversion from the epoched data in Matlab format into our `Data` format, we preprocessed both the training and test data in the following way: First the data was 13Hz low-pass- and 9Hz high-pass-filtered and subsampled to 50Hz. Note that we used the `filtfilt` method here, which implements a non-causal forward-backward filter. This is only feasible in offline analysis where the complete data set is available from the beginning. For online experiments one has to use the `lfilter` method which implements a regular IIR/FIR filter (cf. Section "Performing Online- and Simulated Online Experiments").

```
fs_n = dat.fs / 2
b, a = proc.signal.butter(5, [13 / fs_n],
                          btype='low')
dat = proc.filtfilt(dat, b, a)
b, a = proc.signal.butter(5, [9 / fs_n],
                          btype='high')
dat = proc.filtfilt(dat, b, a)
dat = proc.subsample(dat, 50)
```

After filtering and subsampling, we calculated the Common Spatial Filter (CSP) on the training set:

```
filt, pattern, _ = proc.calculate_csp(dat)
```

In the next step we perform the spatial filtering by applying the CSP filters to the training- and test data to reduce the 64-channel data down to 2 components. `apply_csp` by default uses the first and last spatial filter (i.e. columns of the `filt` argument). If more or other spatial filters are needed one can overwrite the

```
dat = proc.apply_csp(dat, filt)
```

The last step of the preprocessing is creating the feature vectors by computing the variance along the time axis and the logarithm thereof:

```
fv = proc.variance(dat)
fv = proc.logarithm(fv)
```

Until here, the processing of training and test data is almost identical, the only difference being the calculation of the CSP filters on the training set only. In the next steps we will use `fv_train` and `fv_test` instead of `fv` to differentiate between the feature vectors of the training- and test data.

During the preprocessing we reduced the training data with the shape (278, 3000, 64) down to a feature vector with the shape (278, 2) – meaning each trial is represented by two numbers. Analogous, the test data was reduced from (100, 3000, 64) to (100, 2). After the preprocessing of training- and test-data, we can train the Linear Discriminant Analysis (LDA) classifier, using the feature vector of the training data and the class labels:

```
cfy = proc.lda_train(fv_train)
```

Applying the feature vector of the test data to the classifier yields the projection of the test data on the hyperplane, trained by the `lda_train` method:

```
result = proc.lda_apply(fv_test, cfy)
```

The result is an array of LDA classifier outputs (i.e. one per trial), and we use the sign of each element to determine the corresponding class membership for each trial.

Analysis and Results

In Fig. 2 we have visualized two CSP spatial patterns, which were also calculated during the computation of the CSP filter. We show the pattern for the imagined pinky movement (left pattern) as well as for the tongue movement (right pattern). Each pattern is an 8x8 grid, where each cell represents the respective electrode on the ECoG grid. The class-specific activation patterns show the spatially distinct regions that give rise to the strongest ERD/ERS during imagined movement of either the pinky or the tongue. See Haufe et al. (2014) for a discussion about the interpretability of spatial patterns in contrast to spatial filters.

Comparing our resulting predicted labels with the true labels, showed that our method has an accuracy of 94 % for that data set. The expected accuracy if classification is made by chance is 50 %. That result is comparable with the results of the BCI Competition, where the first three winners reached an accuracy of 91 %, 87 %, and 86 %. It is important to note that the goal here was not to "win" the competition, but to provide some context for the results we achieved. We had the advantage of having the true labels, which the competitors of the competition had not.

ERP Component Classification in EEG Recordings

The second data set uses Electroencephalography (EEG) recordings, provided by the Wadsworth Center, NYS Department of Health, USA. The data were acquired using BCI2000's Matrix Speller paradigm (Schalk et al. 2004), originally described in (Donchin et al. 2000). The subject had to focus on one out of 36 different characters, arranged in a 6x6 matrix. The rows and columns were successively and randomly intensified. Two out of 12 intensifications contained the desired character (i.e. one row and one column). The event-related potential (ERP) components evoked by these target stimuli are different from those ERPs evoked by stimuli that did not contain the desired character. The ERPs are composed of a combination of visual and cognitive components (Brunner et al. 2010; Treder and Blankertz 2010).

The subject's task was to focus her/his attention on characters (i.e. one at a time) in a word that was prescribed by the investigator. For each character of the word, the 12 intensifications were repeated 15 times before moving on to the next character. Any specific row or column was intensified 15 times per character and there were in total 180 intensifications per character.

The data was recorded using 64 channel EEG. The 64 channels covered the whole scalp of the subject and were aligned according to the 10-20 system. The collected signals were bandpass filtered from 0.1-60Hz and digitized at 240Hz.

The data set consists of a training set of 85 characters and a test set of 100 characters for each of the two subjects. For the trainings sets the labels of the characters were available. The task for this data set was to predict the labels of the test sets using the training sets and the labels.

After the initial conversion of the original data into our `Data` format, the data was available as continuous data in

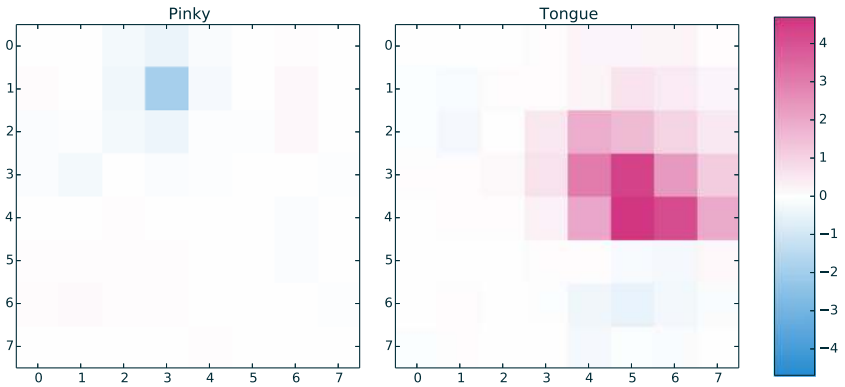

Fig. 2 Spatial activation patterns of CSP components that show the strongest class-discriminative ERD/ERS for imagined pinky or tongue movement as measured on the 8x8 ECoG grid

a [{time, channel] fashion, with the markers describing the positions in the data stream where the intensifications took place.

In the first step, the data was 30Hz low-pass- and 0.4Hz high-pass filtered and subsampled to 60Hz:

```
fs_n = dat.fs / 2

b, a = proc.signal.butter(16, [30 / fs_n],
                          btype='low')
dat = proc.lfilter(dat, b, a)

b, a = proc.signal.butter(5, [.4 / fs_n],
                          btype='high')
dat = proc.lfilter(dat, b, a)

dat = proc.subsample(dat, 60)
```

In contrast to the ECoG data set, which was already in the epoched form, this data set is a continuous recording and has to be segmented into epochs. For segmentation we use the markers which define certain events (MRK_DEF) and "cut" the data around the time point defined by the marker and a segmentation interval (SEG_IVAL), in this case $[0, 700)$ ms around the respective marker onset, and assign each resulting chunk to a class defined by the marker definition. The resulting epoched data has the form [class, time, channel]:

```
epo = proc.segment_dat(dat, MRK_DEF, SEG_IVAL)
```

In order to receive good classification results for ERP classification tasks, it is a good strategy to calculate the means over certain time intervals (JUMPING_MEANS_IVALS) for each channel, instead of using the data set as is. The time intervals are highly subject specific and have to be chosen by using the classwise average, signed r^2 values or some other heuristic (Blankertz et al. 2011). The number of intervals is usually between 3-6.

By appending the average values for each channel to a vector, we receive the feature vectors for each trial:

```
fv = proc.jumping_means(epo, JUMPING_MEANS_IVALS)
fv = proc.create_feature_vectors(fv)
```

Now we can use again the LDA to train a classifier using the feature vectors of the training data and the labels and classify the feature vector of the testing data:

```
cfy = proc.lda_train(fv_train)
lda_out_prob = proc.lda_apply(fv_test, cfy)
```

The result is a LDA classifier output for each trial (i.e. intensification), predicting whether that intensified row or column was the one the subject was concentrating on. In order to get the actual letters the subjects wanted to spell, one has to combine the 15 classifier outputs for each row and column that the row/column has been intensified into one respectively and choose the most probable row and column. Each row-column combination defines a letter which has been the one the subject was probably attending to. This "unscrambling" step has been omitted in this paper for the sake of brevity but is available in the example script.

Analysis and Results

So far we did not explain how we choose the time intervals for the means for each subject. Figure 3 shows the classwise average time course for three selected channels (FCz, Cz, and Oz) and both subjects. Those plots can be generated using plot_timeinterval from Wyrm's plot module. As expected, we see for both subjects an early activation in the occipital areas around 200ms, followed by activation in the central and fronto-central areas. We also see that the kind of activation, especially in the occipital area, differs highly between the two subjects: subject A has a positive response on channel Oz around 200ms whereas subject B

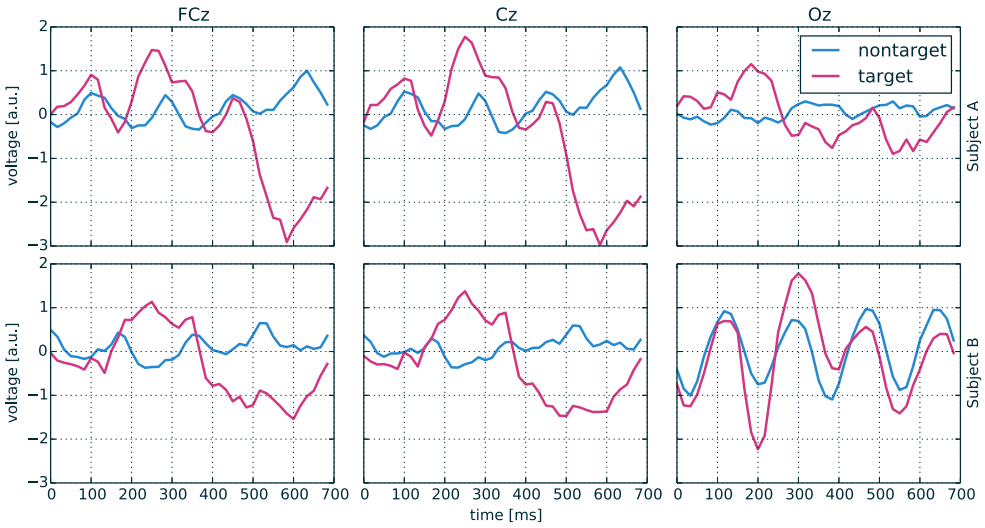

Fig. 3 Classwise average time courses for subject A (*top row*) and subject B (*bottom row*) for three selected channels. The averages were calculated on the whole training set, t=0 ms is the onset of the stimulus

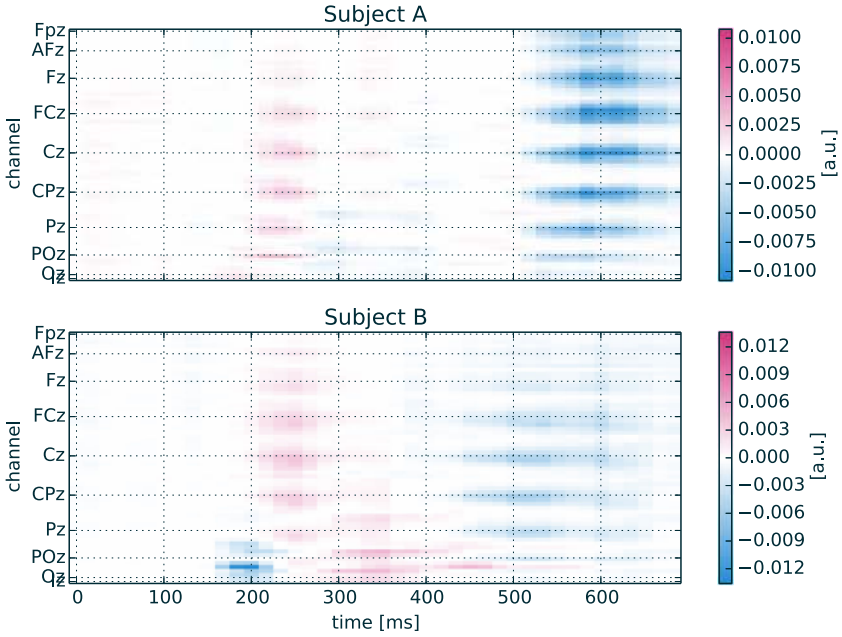

Fig. 4 Signed r^2-values for subject A (*top row*) and subject B (*bottom row*). The channels are sorted from frontal to occipital and within each row from left to right. The blobs show the time intervals for each channel, which discriminate best against the other class.

has a negative one. Moreover subject B's Oz resonates much stronger at the frequency the stimuli were presented with than subject A. Not only is the inter-subject difference very large, also the variance of single time courses compared to the average is especially high for ERP experiments.

In order to quantify the discriminative information for each channel and time point, we compute the signed r^2-values, cf. (Blankertz et al. 2011). Those r^2-values serve as univariate statistical measures for separability. The discriminative information across all channels and time points can then be visualized as a matrix using `plot_spatio_temporal_r2_values` from Wyrm's `plot` module (see Fig. 4).

Comparing the signed r^2-values on Fig. 4 between the two subjects, we see both subjects feature a positive ERP component between 200 and 280ms after stimulus onset. This component is known as P300 component which is strongest in in the central- to frontal areas, as shown in Fig. 3. Moreover, subject B displays a strong negative component (called N200) around 150-250ms after stimulus onset (Fig. 6). This visual N200 component is mainly located in occipital areas.

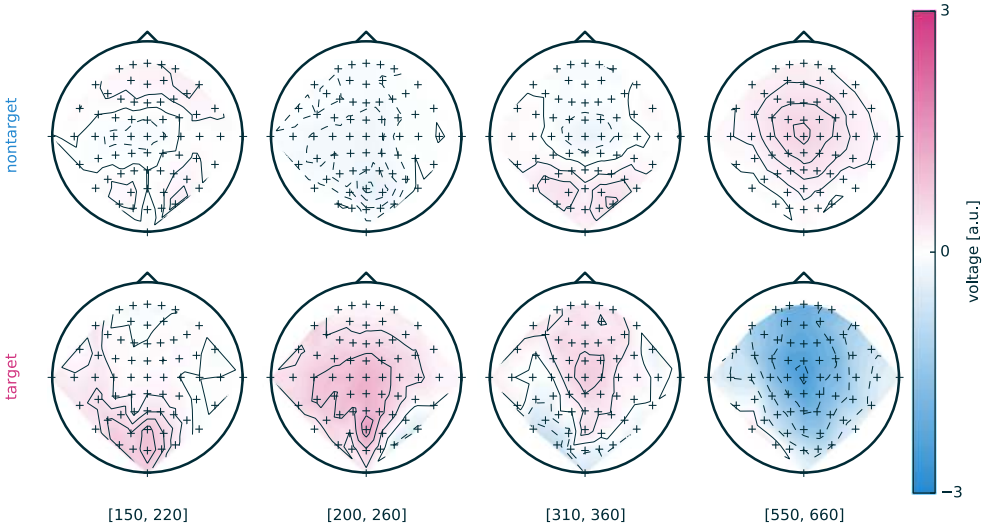

Fig. 5 Spatial topographies of the average voltage distribution for the different time intervals used for classification for subject A. The top row shows the nontarget trials, the bottom row the targets

In order to find the optimal time intervals for classification, we manually chose four intervals where the signed r^2 have their maximum or minimum and the respective other class does not change the sign on one of the other channels. For subject A, the intervals: 150-220 ms, 200-260 ms, 310-360 ms and 550-660 ms have been chosen; for subject B: 150-250 ms, 200-280 ms, 280-380 ms and 480-610 ms. The Figs. 5 and 6 show the spatial topographies of the average voltage distributions in the selected time intervals we chose for classification. Those scalp plots can be generated using `plot_scalp`.

Comparing the resulting letters, predicted by our classification with the real ones the subjects were supposed to spell, our implementation reaches an accuracy of 91,0 % (91 % for both, subject A and subject B) which is comparable with the results of the winners of the competition, where the first three winners reached an accuracy of 96,5 %, 90,5 %, and 90 %. The expected accuracy if classification is made by chance is 2,8 %.

Note that a much better classification can be achieved by a much simpler preprocessing method, namely: 10Hz low-pass filtering the data, subsampling down to 20Hz and just creating the feature vectors (without calculating the means over intervals):

```
B, A = proc.signal.butter(5, [10 / 120], btype='low')
dat = proc.filtfilt(dat, B, A)

dat = proc.subsample(dat, 20)
epo = proc.segment_dat(dat, MRK_DEF, SEG_IVAL)

fv = proc.create_feature_vectors(epo)
```

The results for that classification are 96 % (96 % for subject A and B). Due to the increased dimensionality of features, this approach requires a lot of training examples to work well and which are available in this data set. In practice, one aims at keeping the calibration short such that interval selection as explained above can be expected to work better.

Performing Online- and Simulated Online Experiments

In this section we will show how to use Wyrm to perform an online experiment. To demonstrate the experiment we will use the ERP data set from Section "ERP Component Classification in EEG Recordings", subject A and perform the classification task in an online fashion by using a software amplifier that reads data from a file and returns signals and markers in small chunks in realtime in exactly the same manner as when acquiring data from a real amplifier. This capability of realistically simulating online processing of Wyrm is not only good for demonstration but also for other purposes, see discussion in Secion "Discussion".

The principal processing steps and parameters for filtering, subsampling, etc., that lead to the classification are the same as in the offline experiment shown in Section "ERP Component Classification in EEG Recordings ", so we can focus here on the differences between the offline and online processing.

In order to simulate an online experiment with the available EEG data, we will use the `ReplayAmp` pseudo amplifier from the Mushu signal acquisition (Venthur and Blankertz 2012). The pseudo amplifier can load a complete data set and its `get_data` method returns only as much data and markers as possible given the sampling frequency of the data and the time passed since the last call of `get_data`. From our toolboxes point of view this software amplifier

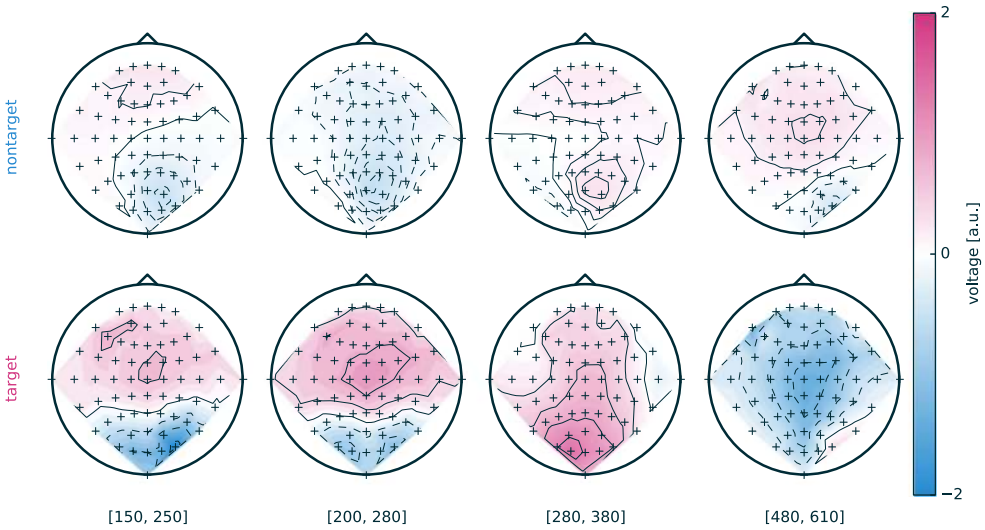

Fig. 6 Spatial topographies of the average voltage distribution for the different time intervals used for classification for subject B

behaves like a real amplifier. Using this `ReplayAmp` also makes the experiment reproducible for the reader as the Mushu signal acquisition, the online experiment script, as well as the data used, are freely available.

In contrast to the offline experiment, where the entire data set is available, in the online setting we have to process the incoming data chunk-wise. The chunks of data typically have a length of just a few samples (or blocks). This leads to differences in some of the processing steps:

When filtering the data chunk-wise, we have to use `lfilter` with filter delay values in order to receive the same results as if we were filtering the whole data set at once.

The subsampling from 240Hz to 60Hz internally works by returning every 4th sample from the data to be subsampled. When subsampling chunk-wise, we have to make sure that the data to be subsampled has a length of multiples of 4 samples in order to avoid losing samples between the chunks of data. For that we have to either set a block size of 4 samples (or an integer multiple of 4) in the amplifier or utilize Wyrm's block buffer. Since most amplifiers allow for a configuration of the block size, we set the block size of 4 samples in the `ReplayAmp` as well.

If the amplifier does not support the configuration of the block size, one can use Wyrm's implementation of a block buffer. The block buffer behaves like a queue, a first-in-first-out data structure, that is unlimited in size. The block buffer has two functions: `append` and `get`. `append` accepts a continuous `Data` object and appends it to its internal data storage. `get` returns (and internally deletes) the largest possible block of data that is divisible by `blocksize`, starting from the beginning of its internal data storage. After a `get`, the block buffer's internal data has at most the length $blocksize - 1$. A subsequent call of `get` would return empty data, a subsequent call of `append` will append the new data to the remaining data in the internal representation and so on.

We will also utilize Wyrm's implementation of a ring buffer where we can `append` small chunks of data in each iteration of the online loop and `get` the last 5000ms of the acquired data to perform the classification on.

Training

The online experiment can be divided into the training part and the online part. In the first part, the training EEG data is recorded and after the recording is done, the entire training data is used for training the LDA classifier, much like in the offline setting. The signal processing and training of the LDA classifier in the training part is identical to the signal processing and training of the LDA in the offline analysis in Section "ERP Component Classification in EEG Recordings".

```
from wyrm import processing as proc
from wyrm import io
from wyrm.types import RingBuffer
import libmushu

cnt = io.load_bcicomp3_ds2(TRAIN_DATA)

fs_n = dat.fs / 2
b, a = proc.signal.butter(5, [30 / fs_n],
                          btype='low')
cnt = proc.lfilter(cnt, b, a)
b, a = proc.signal.butter(5, [.4 / fs_n],
                          btype='high')
cnt = proc.lfilter(cnt, b, a)
cnt = proc.subsample(cnt, 60)

epo = proc.segment_dat(cnt, MARKER_DEF_TRAIN,
                       SEG_IVAL)

fv = proc.jumping_means(epo, JUMPING_MEANS_IVALS)
fv = proc.create_feature_vectors(fv)

cfy = proc.lda_train(fv)
```

Online Classification

In the second part we use the classifier `cfy` obtained from the training, to classify the incoming data.

First we prepare the online loop. We load the test data set and provide it to Mushu's `ReplayAmp`. Note how we configure the amplifier to use a block size of four samples and set it into the realtime mode.

```
cnt = io.load_bcicomp3_ds2(TEST_DATA)
amp = libmushu.get_amp('replayamp')
amp.configure(data=cnt.data, marker=cnt.markers,
              channels=cnt.axes[-1], fs=cnt.fs,
              realtime=True, blocksize_samples=4)
```

Assuming we have an amplifier `amp`, we need to know the sampling frequency, the names of the EEG channels and the number of channels:

```
amp_fs = amp.get_sampling_frequency()
amp_channels = amp.get_channels()
n_channels = len(amp_channels)
```

Then we setup the ring buffer with a length of 5000 ms.

```
rb = RingBuffer(5000)
```

We calculate the filter coefficients and the initial filter states for the low- and high-pass filters and put the amplifier into the recording mode.

```
fn = amp.get_sampling_frequency() / 2
b_low, a_low = proc.signal.butter(5, [30 / fn],
                                  btype='low')
b_high, a_high = proc.signal.butter(5, [.4 / fn],
                                    btype='high')

zi_low = proc.lfilter_zi(b_low, a_low, n_channels)
zi_high = proc.lfilter_zi(b_high, a_high, n_channels)

amp.start()
```

The actual online processing happens in a loop. At the beginning of each iteration we acquire new data from the amplifier and convert it into Wyrm's data format using the `convert_mushu_data` method provided by Wyrm's `io` module.

```
while True:
data, markers = amp.get_data()
cnt = io.convert_mushu_data(data, markers,
amp_fs, amp_channels)
```

The remaining code samples from this section are *all* part of the loop. We removed the first level indentation from the loop for better readability.

We can filter the data using `lfilter` and the optional `zi` parameter that represents the initial conditions for the filter delays. Note how `lfilter` also returns the initial conditions for the next call of `lfilter` when called with the optional `zi` parameter:

```
cnt, zi_low = proc.lfilter(cnt, b_low, a_low,
                           zi=zi_low)
cnt, zi_high = proc.lfilter(cnt, b_high, a_high,
                            zi=zi_high)
```

The filtered data can now be subsampled from the initial 240 Hz to 60 Hz.

```
cnt = proc.subsample(cnt, 60)
```

Now we append the data to the ring buffer and query the ring buffer for the data it contains, thus we will always have the last 5000 ms of acquired data. Before putting the data into the ring buffer we store the number of new samples in a variable as this number is needed later when calculating the epochs.

```
newsamples = cnt.data.shape[0]

rb.append(cnt)
cnt = rb.get()
```

In the next step we segment the 5000 ms of data. Since the difference between the 5000ms of data from *this* iteration and the 5000 ms from the *previous* iteration is probably only a few samples, we have to make sure that `segment` returns each epoch only once within all iterations of the loop in order to avoid classifying the same epoch more than once. For that we provide the `segment` method with the optional `newsamples` parameter. Using the information about the number of new samples, `segment` can calculate which epochs must have already been returned in previous iterations and returns only the new epochs. Note, that `segment` has to take into account, that the interval of interest `SEG_IVAL` typically extends to poststimulus time. I.e., a segment is only returned when enough time has elapsed after a marker in order to extract the specified interval.

```
epo = proc.segment_dat(cnt, MARKER_DEF_TEST,
SEG_IVAL,
newsamples=newsamples)
if not epo:
continue
```

If `segment` does not find any valid epochs, we abort this iteration and start the next one. Otherwise `epo` contains at least one or more epochs. On these epochs we calculate the jumping means, create the feature vectors and apply it to the LDA classifier, exactly as in the offline example.

```
fv = proc.jumping_means(epo, JUMPING_MEANS_IVALS)
fv = proc.create_feature_vectors(fv)

lda_out = proc.lda_apply(fv, cfy)
```

What happens with the output is highly application dependent. In the online experiment example script available in the `examples` directory that contains the complete script from above, we use `lda_out` to calculate the probabilities for each letter after each iteration of the loop. After 12 intensifications we select the most probable letter, reset all probabilities to zero and continue with the next letter. Running the script takes ca 50 minutes (equalling the duration of the recording since we used the setting `realtime=True` in the initialization of the `ReplayAmp`; for other options see the Section "Discussion") and classification accuracy for correctly detected letters is identical with the accuracy of subject A in the offline classification (91 %).

On the testing machine, a Laptop with a quad-core Intel i7 CPU at 2.8 GHz, it takes a fairly constant time of 3.5 ms to complete a full iteration of the main loop. This does not take into account the iterations that are aborted early because of empty epochs, those iterations are naturally completed even faster.

In ERP experiments, incoming data is usually processed with the same frequency as the stimuli are presented. For ERP experiments, 200 ms is a common interval between two stimuli. In this case the time between two stimuli was 175 ms, which is also the maximum time allowed to process the data per iteration. With 3.5 ms, Wyrm processed the data faster by the order of two magnitudes which gives a lot of margin. 3.5 ms is also well below the maximum time of 16.7 ms needed to process the data block by block (if one block consists of 4 samples), and would be still faster than the 4.17 ms needed to processes the data sample by sample, given the sampling frequency of 240 Hz.

Performance

In this section we will investigate further on Wyrm's real-time capabilities and performance limitations. For that we will use the online ERP-experiment from Section "Performing Online- and Simulated Online Experiments" and modify the two parameters that directly influence the size of the data to be processed: the sampling frequency and the channel count. The goal of this analysis is to evaluate the performance of Wyrm with increasing load. Moreover,

we want to assess and describe scenarios in which the performance breaks down.

The code we used to measure the performance is based on the online experiment from the previous section. A few changes were made to increase the load and measure the performance: (1) Instead of real data from an amplifier, we generate random data. The signal generator behaves like a standard amplifier: in each iteration it produces as much data as possible, given the configured sampling frequency and the last time data was acquired. (2) We added a block buffer (Section "Performing Online- and Simulated Online Experiments") that ensures that the data is processed in multiples of 10 ms. The block buffer is usually not necessary if the amplifier supports a configurable block size, and we added it here to increase the load on the computer during the online loop. (3) Instead of subsampling down to 60 Hz, we subsample to 100 Hz. (4) We generate markers every 10 ms, which yields 100 classifications per second.

To assess the performance, we measured the execution times (`dt`) for full iterations of the online loop. The measurements start before the data is generated and end after the classification. For each scenario we measured the execution times of 500 full iterations. We did not measure the execution times of iterations that aborted early due to empty blocks from the block buffer or empty epochs after the segmentation. Since the ring buffer's `append` method is slightly faster when the ring buffer has not been completely filled yet, the measurements start only after the ring buffer has been completely filled in order to avoid the better execution times in the beginning of the measurement. In order to keep up with the incoming data in this scenario, a full iteration should not take longer than 10 ms.

Since the code is based on the code from Section "Performing Online- and Simulated Online Experiments", we show it here only in an abbreviated form, to highlight the measurement method. The full script that measures the timings and generates the plot from Fig. 7, can be found in the `performance.py` script in the `examples` directory.

```
full_iterations = 0
while full_iterations < 500:

t0 = time.time()

# generate data

# blockbuffer (continue)

# filter
# subsample

# ringbuffer

# segmentation (continue)

# feature vectors
# classification

# continue if ringbuffer is not full yet

# time needed to process the new data
dt = time.time() - t0

full_iterations += 1
```

We ran this experiment with three sampling frequencies: 100 Hz, 1k Hz, and 10 kHz, and three channel counts: 50, 100, and 500. This results in 9 combinations of sampling frequencies and channels. The results are shown in Fig. 7 (left). Each box plot displays the 500 measurements (i.e. the execution times of a full iteration of the main loop). As we can see, Wyrm can, in all cases, process new data in less than 10 ms and thus, keep up with the classification rate of 100 classifications per second. We also note that the execution

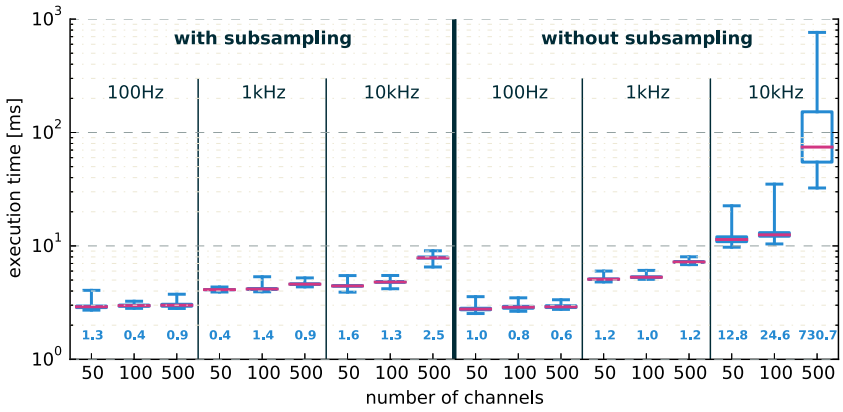

Fig. 7 Execution times of full iterations of the online loop in various settings. On the left side, the data was subsampled to 100 Hz during the processing, on the right no subsampling took place. Each box plot contains 500 measurements, the boxes mark the quartiles, the red lines the medians, and the whiskers the minimum- and maximum values. The blue values below each box show the range between the maximum and minimum value in milliseconds

times are fairly consistent throughout the iterations as the differences between the minimum and maximum times (the blue numbers below each box plot) are in all cases less than 2.6 ms.

To demonstrate Wyrm's limits in online processing, we repeated the experiment and further increased the size of the data to be processed, by omitting the subsampling step (everything else is the same). In this scenario, the second part of the online loop (ring buffer, segmentation, and feature vectors) will have to process 10 and 100 times more data in the 1 kHz and 10 kHz scenarios.

The results are shown in Fig. 7 (right). As we see, in the 100Hz and 1kHz groups the results are still comparable with the *with-subsampling* counterparts, with slightly increased execution times in the 1kHz group and the execution times are still very consistent within each scenario. In the 10kHz group, however, the performance degrades quickly with increasing channel count and Wyrm is unable to keep up with the required 100 classifications per second anymore. With 50 and 100 channels, Wyrm could still process the data in time if we would reduce the requirement from 100 to 10 classifications per second, with 500 channels, the performance breaks down. The execution times themselves also become very unpredictable, ranging more than 700 ms between the best and the worst iteration, and taking almost a second in the worst cases.

We tested Wyrm with two normal sampling frequencies and an extreme value that is rarely used. The same holds for the channel count. We also forced Wyrm to produce a very high classification rate and created a scenario where data is processed without subsampling. We did this deliberately to demonstrate how the performance behaves in both, normal and extreme cases. Whie the performance clearly breaks down in the most extreme case of 10kHz/500 channels/no-subsampling, we also show that Wyrm performs more than sufficient in all other scenarios.

While this experiment does not proof that Wyrm will be fast enough for *all* kinds of BCI experiments, it shows what kind of performance one can expect, given the parameters: sampling frequency, channel count, algorithms used, and classification rate. The experimental setup contained many algorithms that are likely to be used in other scenarios as well. While other experiments might need more expensive operations, they will probably also have more relaxed requirements in at least one other aspect of the experiment (e.g. a reduced number of classifications per second). Other experiments might as well include operations that drastically *reduce* the computation time. For example, in a typical motor imagery experiment, CSP filters are applied, which drastically reduce the number of channels.

Discussion

In the previous sections we showed how to use our toolbox with two very different data sets (ECoG and EEG) and two different paradigms (motor imagery and ERP). For both data sets we provide a brief analysis of key aspects of the data, typical for their respective paradigm. We also demonstrated how to complete the classification tasks, achieving classification accuracies comparable with the ones of the winners of the BCI Competition. This comparison is not meant as a fair competition, since the true labels of the evaluation data have been available to us. The purpose of the comparison was only to provide reproducible evidence that state-of-the-art classification can easily be obtained with our toolbox.

We also showed how to use Wyrm to perform an online experiment. For that we used again the ERP data set and performed the same classification task in an online fashion by replaying the data in realtime using a software amplifier. The online variant yields the exact same result and classification accuracy as the offline classification which demonstrates the consistency of offline and online processing in Wyrm.

Replaying the data in realtime, however, is not a very common use case in BCI as the replay takes as long as the original recording (in this case 50 minutes). We showed it here only to demonstrate the realtime capabilities of our toolbox. Replaying data in timelapse, however, can be useful to evaluate more complex methods, e.g., when some parameters of the feature extraction or the classifiers are continuously adapted. Furthermore, simulated online processing can help the debugging, when online experiments did not work as expected from previous offline test. For that it is desirable to replay the data *faster* than realtime. For that we can turn the realtime mode off so the `ReplayAmp` will always return the next block of data with each call of the `get_data` call. The bigger we set the block size, the faster the data will be processed. Turning the realtime mode in the amplifier off and setting a block size of 40 samples (block length: 166.7 ms), the whole experiment (including loading of the train- and test data sets and training of the classifier) takes a little less than 2 minutes to complete. Changing the block size to 400 samples (block length: 1.7 s), takes less than 50 seconds to complete. All variations of the online experiment yield the exact same results and classification accuracies.

In real online experiments, however, block sizes are typically small. The sampling frequency and channel count influence the size of the data to be processed and thus the performance. We demonstrated that Wyrm performs well in

online experiments, even in extreme scenarios where block sizes are small, and sampling frequency and channel count extremely high.

This shows that Wyrm is not only capable of performing offline and online experiments, but that its functions are written in a way to solve the necessary computations very efficiently.

While Wyrm does not provide a turnkey solution to run BCI experiments, it provides the user with all tools necessary to create online experiments and perform offline analyses. All functions of the toolbox are carefully tested for accuracy and profiled for speed and efficiency.

Conclusion

In this paper we introduced Wyrm, an open source toolbox for BCI. We gave an overview of Wyrm's software architecture and design ideas, and described the fundamental data structure used throughout the toolbox. We also explained how we used unit testing and continuous integration as a mean of quality assurance.

To showcase Wyrm's capabilities, we described in depth the offline analysis and classification of two common BCI paradigms and discussed the results. Furthermore, we demonstrated how to perform an online experiment using Wyrm and showed that Wyrm's functions are efficient enough to process the data in realtime and even faster, if necessary.

As data sets we used publicly available data sets from the BCI Competition III (Blankertz 2005). We also published the scripts explained in this paper along the source code of Wyrm to make the results reproducible for the reader.

Compared to the existing toolboxes in the field of BCI, Wyrm is still very young and other toolboxes may provide a larger set of functions or more sophisticated plotting functions. Some of the other toolboxes are for a special purpose, like SCoT for source connectivity, or BioSig and MNE-Python for analysis of biosignals. In those cases, Wyrm, being a general purpose BCI toolbox, offers a greater scope but at the same time lacks the special features provided by those toolboxes. Toolboxes like FieldTrip, BioSig, MNE-Python and SCoT are only for offline analysis of data, while Wyrm is able to perform offline analyses and online experiments. Regarding the scope of application and features, Wyrm is comparable to the BBCI toolbox and BCILAB. Both toolboxes provide a bigger set of toolbox functions than Wyrm but are otherwise comparable. However, both toolboxes are also written in Matlab and thus depend on commercial software, whereas Wyrm only depends on freesoftware. All toolboxes provide extensive documentation and all, except BioSig and the BBCI toolbox, use unit testing.

We think Wyrm is a valuable addition to the Matlab dominated BCI toolbox ecosystem. Moreover, together with Mushu (Venthur and Blankertz 2012) for signal acquisition and Pyff (Venthur et al. 2010) for feedback and stimulus presentation, we provide a completely free and open source BCI system written in Python that is geared towards researchers that develop new BCI paradigms and- methods.

Information Sharing Statement

All source code and data utilized in this work is freely available. Wyrm's source code can be downloaded at http://github.com/bbci/wyrm. The online reference and documentation is available at http://bbci.github.io/wyrm/. Wyrm is licensed under the terms of the MIT license. The data sets used in this paper are from the BCI Competition III, and available from the homepage (http://bbci.de/competition/iii/). The code that loads the BCI Competition data into Wyrm's native format is included in Wyrm's `io` module. The classification-, analysis-, and online code shown in this paper is available in Wyrm's `examples` directory. The `ReplayAmp` used in the simulated online example is part of the Mushu signal acquisition which is available at http://github.com/bbci/mushu and licensed under the terms of the GPL.

Acknowledgments This work was supported in part by grants of the BMBF: 01GQ0850 and 16SV5839. The research leading to this results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreements 611570 and 609593.

Conflict of interests The authors declare that they have no conflict of interest.

References

Bissyandé, T.F., Thung, F., Lo, D., Jiang, L., & Réveillère, L. (2013). Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *Proceedings of the 37th annual international computer software & amp; applications conference (COMPSAC 2013)* (pp. 1–10). Kyoto. http://hal.archives-ouvertes.fr/hal-00809451.

Blankertz, B. (2005). BCI Competition III results (web page). http://www.bbci.de/competition/iii/results.

Blankertz, B., Müller, K.R., Krusienski, D., Schalk, G., Wolpaw, J.R., Schlögl, A., Pfurtscheller, G., del R Millán, J., Schröder, M., & Birbaumer, N. (2006). The BCI competition III: Validating alternative approachs to actual BCI problems. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, *14*(2), 153–159. doi:10.1109/TNSRE.2006.875642.

Blankertz, B., Tomioka, R., Lemm, S., Kawanabe, M., & Müller KR (2008). Optimizing spatial filters for robust EEG single-trial analysis. *IEEE Signal Processing Magazine*, *25*(1), 41–56. doi:10.1109/MSP.2008.4408441.

Blankertz, B., Lemm, S., Treder, M.S., Haufe, S., & Müller, K.R. (2011). Single-trial analysis and classification of ERP components – a tutorial. *NeuroImage*, *56*, 814–825. doi:10.1016/j.neuroimage.2010.06.048.

Brunner, C., Andreoni, G., Bianchi, L., Blankertz, B., Breitwieser, C., Kanoh, S., Kothe, C., Lécuyer, A., Makeig, S., Mellinger, J., Perego, P., Renard, Y., Schalk, G., Susila, I., Venthur, B., & Müller-Putz, G. (2013). Bci software platforms. In B.Z. Allison, S. Dunne, R. Leeb, J. Del R Millán, & A. Nijholt (Eds.) *Towards practical brain-computer interfaces, biological and medical physics, biomedical engineering*. doi:10.1007/978-3-642-29746-5_16 (pp. 303–331). Berlin: Springer.

Brunner, P., Joshi, S., Briskin, S., Wolpaw, J.R., Bischof, H., & Schalk, G. (2010). Does the "P300" speller depend on eye gaze? *Journal of neural engineering*, *7*, 056,013. doi:10.1088/1741-2560/7/5/056013.

Dähne, S., Meinecke, F.C., Haufe, S., Höhne, J., Tangermann, M., Müller, K.R., & Nikulin, V.V. (2014). SPoC: a novel framework for relating the amplitude of neuronal oscillations to behaviorally relevant parameters. *NeuroImage*, *86*(0), 111–122. doi:10.1016/j.neuroimage.2013.07.079. http://www.sciencedirect.com/science/article/pii/S1053811913008483.

Donchin, E., Spencer, K., & Wijesinghe, R. (2000). The mental prosthesis: assessing the speed of a p300-based brain-computer interface. *IEEE Transactions on Rehabilitation Engineering*, *8*(2), 174–179. doi:10.1109/86.847808.

Goodger, D., & van Rossum, G. (2001). Docstring conventions. URL http://www.python.org/dev/peps/pep-0257/.

Gramfort, A., Luessi, M., Larson, E., Engemann, D.A., Strohmeier, D., Brodbeck, C., Goj, R., Jas, M., Brooks, T., Parkkonen, L., & Hämäläinen, M. (2013). Meg and eeg data analysis with mne-python. *Frontiers in Neuroscience*, *7*(267). doi:10.3389/fnins.2013.00267. http://www.frontiersin.org/brain_imaging_methods/10.3389/fnins.2013.00267/abstract.

Haufe, S., Meinecke, F., Görgen, K., Dähne, S., Haynes, J.D., Blankertz, B., & Bießmann, F. (2014). On the interpretation of weight vectors of linear models in multivariate neuroimaging. *NeuroImage*, *87*, 96–110. doi:10.1016/j.neuroimage.2013.10.067.

Hunter, J. (2007). Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, *9*(3), 90–95. doi:10.1109/MCSE.2007.55.

Jones, E., Oliphant, T., Peterson, P., et al. (2001). SciPy: Open source scientific tools for Python. URL http://www.scipy.org/.

Kothe, C.A., & Makeig, S. (2013). BCILAB: a platform for brain–computer interface development. *Journal of neural engineering*, *10*(5), 056,014.

Krell, M.M., Straube, S., Seeland, A., Wöhrle, H., Teiwes, J., Metzen, J.H., Kirchner, E.A., & Kirchner, F. (2013). pyspace– signal processing and classification environment in python. Frontiers in Neuroinformatics 7.

Lal, T.N., Hinterberger, T., Widman, G., Schröder, M., Hill, N.J., Rosenstiel, W., Elger, C.E., Schölkopf, B., & Birbaumer, N. (2005). Methods towards invasive human brain computer interfaces. In L.K. Saul, Y. Weiss, & L. Bottou (Eds.) *Advances in neural information processing systems*, (Vol. 17 pp. 737–744). Cambridge: MIT Press.

Lipow, M. (1982). Number of faults per line of code. *IEEE Transactions on SE– Software Engineering*, *8*(4), 437–439. doi:10.1109/TSE.1982.235579.

Louden, K. et al. (2011). Programming languages: principles and practices. Cengage Learning.

McKinney, W. (2012). Python for Data Analysis: Data Wrangling with PandasNumPy, and IPython. O'Reilly Media.

Oliphant, T.E. (2007). Python for scientific computing. *Computing in Science Engineering*, *9*(3), 10–20. doi:10.1109/MCSE.2007.58.

Oostenveld, R., Fries, P., Maris, E., & Schoffelen, J.M. (2011). Fieldtrip: open source software for advanced analysis of meg, eeg, and invasive electrophysiological data. *Computational intelligence and neuroscience*, *2011*, 1.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, *12*, 2825–2830.

Pérez, F., & Granger, B. (2007). Ipython: A system for interactive scientific computing. *Computing in Science Engineering*, *9*(3), 21–29. doi:10.1109/MCSE.2007.53.

Ramoser, H., Muller-Gerking, J., & Pfurtscheller, G. (2000). Optimal spatial filtering of single trial eeg during imagined hand movement. *IEEE Transactions on Rehabilitation Engineering*, *8*(4), 441–446.

Renard, Y., Lotte, F., Gibert, G., Congedo, M., Maby, E., Delannoy, V., Bertrand, O., & Lécuyer, A. (2010). Openvibe: an open-source software platform to design, test, and use brain-computer interfaces in real and virtual environments. *Presence: teleoperators and virtual environments*, *19*(1), 35–53.

Schalk, G., McFarland, D.J., Hinterberger, T., Birbaumer, N., & Wolpaw, J.R. (2004). BCI2000: a general-purpose brain-computer interface (BCI) system. *IEEE Transactions on Biomedical Engineering*, *51*(6), 1034–1043.

Schlögl, A., & Brunner, C. (2008). Biosig: a free and open source software library for BCI research. *Computer*, *41*(10), 44–50.

Schreiner, T., Hill, N., Schreiner, T., Puzicha, C., & Farquhar, J. (2008). *Development and application of a python scripting framework for bci2000*. Tübingen: Master's thesis Universität Tübingen.

Treder, M.S., & Blankertz, B. (2010). (C)overt attention and visual speller design in an ERP-based brain-computer interface. *Behavioral and Brain Functions*, *6*, 28. http://www.behavioralandbrainfunctions.com/content/6/1/28.

Venthur, B., & Blankertz, B. (2012). Mushu, a free-and open source BCI signal acquisition, written in python. In *Engineering in Medicine and Biology Society (EMBC), 2012 Annual International Conference of the IEEE*. doi:10.1109/EMBC.2012.6346296, (Vol. 2012 pp. 1786–1788): IEEE.

Venthur, B., Scholler, S., Williamson, J., Dähne, S., Treder, M.S., Kramarek, M.T., Müller KR, & Blankertz, B. (2010). Pyff—a pythonic framework for feedback applications and stimulus presentation in neuroscience. *Frontiers in Neuroinformatics*, *4*, 100. doi:10.3389/fninf.2010.00100. http://www.frontiersin.org/neuroinformatics/10.3389/fninf.2010.00100/abstract.