STRUCTURAL IMPROVEMENTS TO DATA PROCESSING FOR
BRAIN-COMPUTER INTERFACES

A THESIS SUBMITTED TO THE GRADUATE DIVISION OF THE
UNIVERSITY OF HAWAI‘I AT MĀNOA IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

ELECTRICAL ENGINEERING

MAY 2015

By
Trevor Alexander

Thesis Committee:
Anthony Kuh, Chairperson
Tomasz M. Rutkowski
Narayana Prasad Santhanam
5 Optimizations

5.1 Memory Optimizations
5.1.1 Strict call-by-reference

5.2 Processing Optimizations
5.2.1 Explicit parallelism

6 Benchmark Results

6.1 Memory
6.2 Results
6.3 Processing
6.4 Results
6.4.1 Filtering
6.4.2 SDA
6.4.3 SVM
6.5 Expected speedup in real processing

7 Summary

7.1 Main Results
7.2 Summary of performance improvements
7.3 Summary of software toolkit features
7.4 Future Work

8 Appendix: Source code

8.1 Benchmarks
8.1.1 filtfiltbenchmark.R
8.1.2 hyperoptbenchmark.R
8.1.3 memorybenchmarks.R

8.2 rBCI
8.2.1 apply_filter.R
8.2.2 classifiers.R
8.2.3 dependencies.R
8.2.4 downsample_dt.R
8.2.5 explorers.R
8.2.6 importers.R
8.2.7 init_backend.R
8.2.8 misc.R
8.2.9 partition_table.R
8.2.10 plot_filter.R
8.2.11 previewers.R
8.2.12 reporter.R
8.2.13 simple_filter.R
8.2.14 structures.R
8.2.15 transformers.R
8.2.16 class_interface.R
8.2.17 class_interface_bayes.R
8.2.18 class_interface_sda.R
8.2.19 class_interface_svm.R
8.2.20 explore_interface.R
| 8.2.21 | explore_interface_anotate.R | 121 |
| 8.2.22 | explore_interface_hist.R | 126 |
| 8.2.23 | explore_interface_means.R | 127 |
| 8.2.24 | explore_interface_partition.R | 129 |
| 8.2.25 | explore_interface_summary.R | 132 |
| 8.2.26 | filter_interface.R | 133 |
| 8.2.27 | filter_interface_downsample.R | 134 |
| 8.2.28 | filter_interface_simple.R | 136 |
| 8.2.29 | import_interface.R | 140 |
| 8.2.30 | import_interface_bci2000.R | 141 |
| 8.2.31 | import_interface_csv.R | 144 |
| 8.2.32 | import_interface_matlab.R | 147 |
| 8.2.33 | import_interface_matlab_type2.R | 150 |
| 8.2.34 | import_interface_rdata.R | 153 |
| 8.2.35 | init_interface.R | 156 |
| 8.2.36 | opts_interface.R | 158 |
| 8.2.37 | opts_interface_multicore.R | 159 |
| 8.2.38 | opts_interface_redis.R | 160 |
| 8.2.39 | opts_interface_snow.R | 161 |
| 8.2.40 | report_interface.R | 163 |
| 8.2.41 | report_interface_graphics.R | 166 |
| 8.2.42 | report_interface_script.R | 166 |
| 8.2.43 | report_interface_text.R | 167 |
| 8.2.44 | tool_interface.R | 167 |
| 8.2.45 | trans_interface.R | 171 |
| 8.2.46 | trans_interface_csp.R | 172 |
| 8.2.47 | trans_interface_kmeans.R | 176 |
| 8.2.48 | trans_interface_pca.R | 179 |

**J Appendix: List of Acronyms**

**L References**
1 Introduction

1.1 Acknowledgements

This work would not be possible without the mentorship, guidance, and kind encouragement of Dr. Tomasz M. Rutkowski of the BCI-lab group, of the Life Science Center of the Tsukuba Advanced Research Alliance, at the University of Tsukuba. Nor would it have advanced without the friendly assistance of the student members of the BCI-lab group, whose hospitable attitude and enthusiasm were infectious.

Most importantly, this thesis would not have advanced without the help of my thesis committee, consisting of Professor Tony Kuh (University of Hawai‘i), Professor Narayana Santhanam (University of Hawai‘i), and Dr. Tomasz Rutkowski (University of Tsukuba). The diversity of their perspectives and specialties were indispensable to completing this research.

1.2 Motivation

The field of brain-computer interfaces is a relatively young one (less than 60 years). It is also an interdisciplinary one, relying on work by specialists in fields from clinical neuroscience to electrical engineering. Most importantly, it has the potential to create a kind of equality of expression for all individuals, since brain-computer interfaces rely, as a group, solely on the mind and brain of an individual, regardless of disability.

The field of free and open-source software, especially for academic research, is also young, but growing fast. Most notably, the functional statistical analysis language R, first created in 1993, has proven to be a fertile ground for machine learning and (applied) statistics researchers. Originally a free implementation of the S language developed at Bell Labs, its collaborative but rigorous culture and fundamentally interdisciplinary user base make it an extremely strong research tool.

Combining these two areas to create a new contribution to the field thus seemed worthwhile at the outset, and natural in retrospect. It is the hope of the author that both of these areas continue to grow and cross-pollinate for the greater good.

1.3 Section summary

In order to place this work in context, we begin in Chapter 2 with a short description of initial work in electroencephalography (EEG) and the first brain-computer interfaces (BCI). The most relevant events are presented in a historical timeline. Current applications are discussed in the context of experimental equipment and common interface paradigms. Concurrent work and applications are introduced, with a critical view of their strengths and weaknesses.

Following that, we get into a detailed description of the data processing involved in EEG-BCI research, in Chapter 3. The process is broken into specific, standard steps with examples, which naturally inform the design of a program to automate them. Potential areas for optimization are briefly mentioned where applicable.

With the above in mind, we next introduce in Chapter 4 the main contribution of this thesis, rBCI, a cross-platform GUI-based research toolkit for BCI research that accelerates and simplifies offline data processing, as well as addressing some of the issues named in concurrent work. Its components are briefly described, and then illustrated through selected use cases.

In Chapter 5, we discuss optimizations applicable to the above data processing, broadly separated into optimizations that reduce memory use, and optimizations that make processing more efficient. In the memory section, we describe the overall strategy for reducing memory overhead and describe how we achieve it through the use of data.table, a high-performance data structure library. Next, we discuss optimizations to the processing itself. We discuss both explicit and implicit parallelism. We make specific mention of the modular parallelism implemented in rBCI, and introduce a number of backends appropriate for high-performance computing (HPC), including local and distributed computing grids. Benchmarks are presented in Chapter 6 that demonstrate significant performance increases.
Finally, a summary of the main results is given in Chapter 7, and future work is discussed. Appendices containing source code used in the thesis, as well as works cited, follow the summary.
2 Background: EEG and brain-computer interfaces

2.1 Historical timeline

The first recorded electroencephalogram was performed by Vladmir Pravdich-Neminsky in 1912 on a canine subject [96]. Twelve years later, in 1924, Hans Berger recorded the first human EEG, giving it its current name [49]. Clinical electroencephalography did not begin in earnest until 1934-35, when Fisher and Lowenback, and Gibbs, Davis and Lennox published studies describing and evaluating characteristic EEG spikes that occur throughout epileptic seizures [84].

While clinical encephalography continued throughout the twentieth century, the first brain-machine interfaces did not appear until 1969 using animal subjects, in experiments on conditioning of cortical activity [41]. The first EEG-based BCI was described by Donchin and Farwell [37] in the 1980s. Since the late 1990s, research has expanded wildly into a number of fields, from military use to neurogaming, with research into both invasive (i.e., cortically implanted) and non-invasive (wearable) interfaces receiving significant funding [51] [59] [141] [31] [73].

2.2 Experimental equipment

A wide variety of EEG-BCI recording equipment is available, with quality strongly dependent on price. A table of popular suites, typical price, and intended use is given below [23] [35] [118] [123] [20].

<table>
<thead>
<tr>
<th>Brand</th>
<th>Approx. price in USD (equivalent configuration)</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Emotiv</td>
<td>799</td>
<td>Gaming/development</td>
</tr>
<tr>
<td>g.tec</td>
<td>&gt; 10000</td>
<td>Research</td>
</tr>
<tr>
<td>OpenBCI</td>
<td>750</td>
<td>Development</td>
</tr>
<tr>
<td>OpenEEG</td>
<td>400</td>
<td>Development</td>
</tr>
<tr>
<td>BioSemi</td>
<td>11000</td>
<td>Research/development</td>
</tr>
</tbody>
</table>

2.3 Common interface paradigms

EEG-BCI paradigms may be broadly divided into two types: structured, and unstructured. An unstructured interface is controllable by the user at any time; it responds to spontaneous user input. A structured interface either requires specific timing or interactions involving both the user and the interface in order to elicit the appropriate input from the user’s EEG. The following sections describe commonly studied EEG features and BCI paradigms that make use of them.

2.3.1 Structured

The majority of BCIs present in the literature rely on a structured presentation of stimulus to function. In this case, it is the user’s EEG response causally linked to a particular stimulus that serves as a sign of user intent [136].

1. ERP

The ERP, or event-related potential, is a time-localized signal that appears in the EEG causally linked to an event [70]. ERPs are typically labeled with a ‘P’ or ‘N’, indicating whether the signal is (primarily) positive- or negative-going, and a number indicating its time offset from the event in question. A brief list of major ERPs used in BCI research is given below.
ERP Theorized purpose
- N100: Control or attention to presented visual/auditory stimulus
- N200: Conscious attention to presented (visual) stimulus
- P200: Reduction of visual search space
- P300: Memory processing and comparison

(a) P300 oddball paradigm
As suggested by neuroscientists in literature, the P300 appears to be involved in memory processing. A common experiment that demonstrates this is known as the oddball paradigm, which is performed in the following way:

i. A stimulus of interest to the subject (the subject may be instructed to focus on a particular stimulus in advance) is embedded with low frequency in a random sequence of uninteresting stimuli (these may be auditory, visual, or even tactile) presented to the subject.

ii. The subject’s EEG is monitored for changes (typically between 250-500ms post-stimulus), specifically, for a positive-going wave.
   A. If a positive-going wave is observed post-stimulus, conclude the subject recognizes the stimulus or a portion of the stimulus.
   B. If no positive-going wave is observed, conclude the subject does not recognize or has no interest in the stimulus.

iii. Results are dramatically improved when the above steps are first performed in a training phase with known outputs. (The training phase is typically used to tune a supervised learning algorithm or preprocessor on known inputs.)

2. SSVEP
The SSVEP, or steady-state visually evoked potential, is an oscillation observed in the V1 matched in frequency (with attenuated higher harmonics) to a visual oscillation perceived by the user. For example, an LED light flashing at 7.2Hz will produce an oscillation in the EEG also at 7.2Hz (and attenuated oscillations at higher harmonics). Thus, an interface may be constructed by supplying multiple stimuli (such as LEDs) at different frequencies in a location where the user can fixate on one of them long enough for a detection algorithm to identify the frequency of interest.

2.3.2 Unstructured
1. ERD
ERD, or event-related desynchronization, is a phenomenon known to occur in the sensorimotor cortex in which a characteristic oscillation (between 7-13Hz, known as the / band) of neurons linked to movement is attenuated upon visualization of (or actually executed) motor activity. In this case, the interface measures the power spectrum at specific positions throughout the motor cortex (after appropriate signal orthogonalization), and chooses a hypothesis based on the area being attenuated.

1 The visual center located in the occipital lobe.
2. Others

Where focused user intent is not the main goal of the interface, other measures can also be employed. Notable measures of user activity include:

- Concentration/fatigue
- Relaxation/arousal
- Engagement/disengagement

Each of these measures is constructed using relatively simple operations on spectral features of the data, typically bands identified by neuroscience as physiologically linked to the phenomena of interest (alpha, beta, gamma, theta). In other words, they are not presupposed to be strongly spatially or temporally localized, and algorithms designed to operate on these features are often posed as detectors.
2.4 Applications

As pointed out by Wolpaw and Wolpaw [136], current BCI design paradigms reuse existing features of the human brain that arise as a natural consequence of its processing activity. In other words, current BCIs do not modify or augment the faculties of the brain, they merely coexist with them. This follows because the brain's biological function is to receive stimuli from the sensory organs of the human body, analyze them, and respond by sending response signals back into the body.

BCI development is driven by three interdisciplinary needs [136]:

- The need for powerful yet inexpensive hardware and software to support complex BCI signal analysis, particularly in realtime
- The continuing need for a greater understanding of the central nervous system
- A "new recognition of the needs and abilities of people disabled" by neurological disorders such as palsy, spinal injuries, stroke, ALS, and many others

The third driver, the need for systems that are "clinically useful", however, is in tension with the profit requirements for proprietary hardware and software. Given the current modest capabilities of BCIs (with most interfaces below 50bits/min as of this writing), the market is limited in this regard.

The most appropriate and successful application for BCIs has been the severely disabled, most notably, sufferers of locked-in syndrome [19], a condition of near-total paralysis. In this situation, a patient cannot utilize the normal outputs of the human brain, but can still perceive certain stimuli. Thus, BCIs are feasible assistance devices in this scenario.

A survey conducted in 2004 of 347 tetraplegics and 334 paraplegics [12] reported the following functional needs, in order of decreasing priority:

<table>
<thead>
<tr>
<th>Paraplegics</th>
<th>Percentage (ranked)</th>
</tr>
</thead>
<tbody>
<tr>
<td>arm and hand function</td>
<td>48.7%</td>
</tr>
<tr>
<td>sexual function</td>
<td>13%</td>
</tr>
<tr>
<td>trunk strength/balance</td>
<td>11.5%</td>
</tr>
<tr>
<td>bladder/bowel function and autonomic regulation</td>
<td>8.9%</td>
</tr>
<tr>
<td>walking movement</td>
<td>7.8%</td>
</tr>
<tr>
<td>normal sensation</td>
<td>6.1%</td>
</tr>
<tr>
<td>relieving chronic pain</td>
<td>4%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tetraplegics</th>
<th>Percentage (ranked)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sexual function</td>
<td>26.7%</td>
</tr>
<tr>
<td>bladder/bowel function and autonomic regulation</td>
<td>18%</td>
</tr>
<tr>
<td>trunk strength/balance</td>
<td>16.5%</td>
</tr>
<tr>
<td>walking movement</td>
<td>15.9%</td>
</tr>
<tr>
<td>relieving chronic pain</td>
<td>12%</td>
</tr>
<tr>
<td>normal sensation</td>
<td>6.1%</td>
</tr>
<tr>
<td>arm and hand function</td>
<td>3.4%</td>
</tr>
</tbody>
</table>

Future applications may also include the enhancement of natural output (such as self-monitoring devices), augmentations (additional arms, digits, or sensory inputs), though these areas are still very much in their infancy.

2.4.1 Applications of ERP-based interfaces

The initial primary application for the severely disabled, in this case, was the P300 speller. P300 spellers are typically one of many variations on this theme:
1. A grid of symbols is displayed before the subject, each darkened slightly.

2. Symbols are lit randomly for a fixed period of time (or number of intensifications, as the literature terms them) during which a single symbol, row, column, or other subset of the grid is illuminated.

3. The subject is instructed to focus or mentally count when the symbol of interest is illuminated.

4. After the set period (or if the algorithm is sufficiently confident in its choice of hypothesis of the desired symbol), the algorithm displays the chosen symbol on the screen.

5. Generalizations exist using Dasher or other semi-predictive algorithms to accelerate processing and allow whole word choice.

An application proposed and evangelized by Farwell, on the other hand, is as a method of interrogation. The oddball paradigm was used by Farwell in conjunction with the FBI as a method of determining whether or not a criminal defendant had 'special knowledge,' i.e. knowledge that only the perpetrator would have, of a crime. Farwell’s system met with apparent success. It was later used as testimonial evidence in a number of court cases, of which some was exculpatory.

Another application, recently introduced by DARPA, uses the P300 response as a partial replacement for alert sentries. A subject fitted with EEG measurement equipment sits comfortably indoors, viewing a visual surveillance feed of an area of interest. The subject is instructed to watch for suspicious movement or activity in the feed. Upon the detection of a P300 response, armed response is automatically dispatched. This is intended to relieve the need for conscious judgement and continual physical presence of sentries.

2.4.2 Applications of SSVEP-based interfaces

As SSVEP allows the user to spontaneously initiate input by choosing to perceive the oscillatory stimulus, it is most suitable for a small number of input choices in a static menu (i.e. one with a single set of options with the user and interface in a fixed location). One example of this is motor control for an electric wheelchair. A console with 4 LEDs flashing at coprime frequencies (to avoid harmonic interference) is placed within the user’s visual field, attached to the wheelchair. The subject chooses to move forward, turn left, turn right, or back up by gazing at the LED associated with that control input.

2.4.3 Applications of ERD-based interfaces

Since the motor cortex is actively used by ambulatory subjects, it can only be used as an interface channel by subjects who cannot otherwise naturally make use of some or all of their motor cortex. Research has shown that motor imagery is sufficient to cause ERD even in the partially paralyzed. As of this writing, decoding of the motor signal at the EEG level is not sufficient enough to allow for detailed motor movements, only binary inputs.

2.5 Concurrent work

This thesis is concerned with the processing of EEG-BCI data as a whole; i.e., improving the process across the board through standardization, optimization, and liberation of tools and routines used. Existing offerings are reviewed here to provide background to the larger issue of dissemination of BCI as a technology.

2.5.1 BCILAB

BCILAB was first released in 2010 by Christian Kothe at the Swartz Center for Computational Neuroscience. It was developed to standardize and simplify BCI research tasks. Despite its relatively recent release, it is one of the most mature toolboxes available for this research.

Its implementation is organized (according to the developer) around a hierarchy of simplicity:
1. Top layer
   - GUI
   - Scripting
   - API

2. Middle layer
   - Model learning
   - Execution
   - Evaluation

3. Lower layer
   - BCI paradigms (template approaches)

BCILAB itself is a plugin to EEGLAB, a more general MATLAB toolbox. Not all of EEGLAB’s capabilities
are directed at (nor appropriate for) BCI research [106], so it is not reviewed here. EEGLAB is available under
the same terms as BCILAB (GPLv3) [29].

2.5.2 SIFT

SIFT, or "Source Information Flow Toolbox", was developed [82] by researchers at the Swartz Center for
Computational Neuroscience. It is intended for use in connective models of EEG data, particularly those that
make use of Granger causality. Its four submodules specifically offer the following tools for (offline) analysis:

1. Preprocessor
   - Normalization
   - Downsampling
   - Detrending

2. Modeler
   - Connectivity estimation
   - Adaptive MVAR modeling
   - Measure estimation on model

3. Statistical Analysis
   - Surrogate statistics (over measures)
   - Analytic statistics

4. Visualization
   - Interactive visualization
   - Graph-theoretic measures
   - Anatomical sourcing

Each of these modules is controllable through a GUI, and dependent on MATLAB to function.
According to literature from this year [142], there is some degree of interoperability between BCILAB and
SIFT, although SIFT is still limited to offline analysis.
2.5.3 **FieldTrip**

FieldTrip is a free/open-source (GPLv3 licensed) toolbox for MATLAB, primarily used for spatial analysis of electrophysiological data, including MEG (magnetoencephalograms) and EEG. Its main uses are modeling and analysis, specifically source reconstruction and statistical testing.

2.5.4 **BioSig**

BioSig was developed by Alois Schloegl at Graz University (now at IST Austria) in 2004. It is essentially a collection of tightly related software scripts written in MATLAB, C, and C++ (more recently, a partial C-to-Python port has been in development), without a unifying GUI. It is available under the GNU Public License, version 3.

2.5.5 **Smario**

Smario is relatively new (on academic scales), having first appeared in print in 2009. Although it is "yet another MATLAB toolbox" for the same (offline) analysis tasks, its focus on modularity and portability of processing tasks is promising. Specifically, scripts assembled from modular components in GUI can be saved and re-used on any other data or system, as long as the appropriate inputs are available. Unfortunately, the project site is empty as of this writing, and the software does not appear in search results.

Figure 2: Smario’s graphical script assembly window. These ordered sequencies are referred to as ‘pipes’ by its authors.

2.5.6 **BCI2000**

BCI2000 is a toolbox for real-time capture and processing of EEG-BCI data, produced by the Schalk Lab. It is primarily aimed at experimentalists and interface designers, although it can export data in a wide variety of formats, including CSV, EDF, and certain MATLAB binary formats. It is written primarily in C++, and source code is freely available for educational or research use.
Although the source is available, compilation is only supported under Windows (Borland C++ Builder 6, Visual Studio, and MinGW) as of this writing. Filters and other functional components may be compiled for use in other programs, and scripts (MATLAB) may be run from within the program. Most notably, BCI2000 can export signals as it receives or processes them through UDP, making it somewhat platform-independent, while still providing online processing capability [3].

2.5.7 Issues

Of the toolboxes introduced here, BioSig is the only tool that does not require either MATLAB or Octave to run. Worse, graphical interfaces provided rely on proprietary MATLAB libraries, rendering many of the included functions useless for those not specialized in programming for this language or application. Despite the good intentions of toolbox developers to make their work open source, this strong dependency on MATLAB has several ramifications. The initial cost is prohibitive to entering researchers and experimental clinical providers [2], commercial developers (i.e., those in Emotiv’s ecosystem), and community efforts (OpenBCI, OpenEEG) [121] [122]. The proprietary licensing surrounding large portions of the toolboxes does not just limit what can be used, but also exerts a negative (i.e. chilling) effect—potential users and developers without the ability to absorb the associated legal and financial risk will avoid these tools.

1. Dissemination of BCI technology

These issues exist against a larger backdrop of barriers to the dissemination of BCI technology as a whole. Wolpaw and Wolpaw name the following major difficulties [136]:

- Regulatory acceptance for profit-seeking enterprise is too costly and difficult (FDA/ISO/Medicare)
- Incentives such as SBIR (Small Business Innovation Research) grants are vulnerable to gaming by so-called ‘SBIR-mills’
- The market demographic segment for current ‘slow’ BCI is too small with respect to total cost-to-market, making BCI an orphan technology

In other words, profit-seeking vehicles cannot disseminate BCI technology effectively to potential customers; at least, not directly. Research programs involving clinical test subjects cite similar difficulties that directly affect beneficiaries:

- Research labs are, as a rule, not equipped to distribute equipment to the number of volunteers they receive for clinical testing [136]
- Proprietary equipment and software are encumbered by expensive licensing barriers [2]
- Research projects are funded to support high-maintenance, short-term projects, ultimately leading to the retraction of any beneficial technology disseminated to clinical subjects, regardless of their success (notably violating the principle of non-abandonment [97]) or the needs of patients [136]

Thus, one can say that BCI research and development, whether engaged in by academic groups or profit-seeking entities, is prevented by structural barriers from sustained engagement with and dissemination to its target populations.

2. Free software

One argument that might be raised in support of "MATLAB-compatible" freely available tools is the existence of GNU Octave, a free alternative to MATLAB. However, Octave is consistently slower than MATLAB [113] at identical tasks, and toolbox support is spotty at best (particularly in the areas of...

---

2 (We do not include bci2000 or FieldTrip here, as their intended uses do not overlap with the tools introduced in this thesis.)
GUI and optimized execution, which are needed in interdisciplinary computational fields like BCI). Thus, making one’s own code available for free use does not address the root problem.

Although each of the introduced software packages have strengths and weaknesses, they compensate for each other—as long as the user remains within the same MATLAB interpreter space. (BioSig, uniquely, does not interoperate with other MATLAB toolboxes, official or otherwise, which makes it very difficult to take advantage of any other innovations outside of its project domain \[110\].) That requirement works against the benefits granted by the above toolkits across the board, especially those that are intended to be monolithic or unified in nature (BCILAB and its related products). Should a research lab or individual researcher commit to a toolbox on this brittle platform, whatever person-hours invested are locked in the code produced that relies on this stack. Avoiding this requires a minimum of effort on both the toolbox developers and the language community.

The case of Smario is illustrative of the responsibilities of package developers to the community, which are arguably greater than that of a proprietary developer working on a single subcomponent of a project. Beyond initial release, the developers also must:

- Ensure code builds and runs on all target machines
- Provide a public system for feedback and bug reports
- Provide an infrastructure for contributions and credit
- Provide a stable location for complete documentation
- Pass any other tests required to deploy to the chosen platform

None of these requirements are set in stone, but they are essential for the health of the project. On the other hand, the community must:

- Follow and consult provided documentation
- Give complete feedback with instructions to reproduce bugs
- Volunteer solutions with proposals
- Stay in sync electronically (and mentally) with package updates

If any of these items are neglected, the project will begin to stagnate. BCI2000, for example, uses only an old forum system \[105\] for bug reports, with informal tracking through their wiki page if there is enough interest. This makes it difficult for developers to track changes or assign responsibility for specific bugs, or even tell what fixes have worked. It also demands continuing expenditure of effort by the user reporting a bug, since they have to separately track discussion and possible solutions from their side as well.

Unfortunately, MATLAB does not have an ecosystem for vetting or validating community code. A "File Exchange" website exists for hosting submitted code \[74\], but basic features of language-centered free/open-source community repositories like CRAN, CTAN, or CPAN \[52\] \[53\] \[111\] such as tight version sync, automated build testing, or user/maintainer bug tracking are not available.

2.5.8 Proposed solutions

We propose and have implemented a solution to many of these issues based entirely on freely available code, while still maintaining many of the advantages of other toolboxes above.

Specifically, we have developed a toolkit, rBCI, which possesses the following desirable characteristics:

- All code free and open-source
• Written entirely in R\(^3\), a free and open-source language \(^{[38]}\) used extensively for data analytics and machine learning
• Interoperable with MATLAB scripts for processing (successfully used for neuroimaging research \(^{[77]}\))
• Supported by a robust, mixed academic/industry community, which has produced both tutorial papers and interoperable packages under peer review since 1996 \(^{[102]}\) \(^{[104]}\) \(^{[14]}\)
• Equipped with powerful modular visualization \(^{[130]}\) and high-performance computing libraries \(^{[50]}\) used to accelerate and simplify BCI processing tasks
• Presented using a cross-platform GUI (GTK \(^{[63]}\)) aimed at a broad range of users, from researchers to hardware hackers

The advantages of implementation in a robust, fertile development environment are many. Examined along with the disadvantages raised in the previous section, it is clear that there exists an unmet need for software that is more robust, agile, and freely available to the increasingly broad community of BCI users, researchers, and developers. rBCI was developed in order to resolve the software barriers to BCI dissemination. Combined with the open hardware initiatives of OpenBCI \(^{[121]}\) and OpenEEG \(^{[122]}\), the result is a completely free toolchain that can be combined with published algorithms and code to create competitive BCIs.

This allows BCI to be freely disseminated as part of a spectrum of assistive technology. While the ethical burdens (for example, informed consent) are no lighter nor less relevant, the low-risk nature of EEG measurement\(^4\) and fait accompli of EEG/BCI hardware production and distribution neatly avoid the massive barriers of international regulation targeted at fixed technologies, which have stymied a continually changing field with ‘too few customers’.

---

\(^3\)Most R components are implemented in C under the hood, making them quite fast.

\(^4\)The sole physical danger presented directly by BCIs themselves is the differential amplifier used in active electrode systems. This itself is a well-known design for creating a reference ground on human skin, and not unique to BCI \(^{[120]}\).
3 Description of data processing

The recording method, equipment setup, and experiment design for EEG-based BCI is often quite uniform. Likewise, the processing steps that follow can be divided into a series of typical stages, which we describe here, with references to features included in the program modules, discussed in detail later.

3.1 Data Processing Steps

![Block diagram illustrating the flow of data through a BCI][135]. Data is recorded from electrodes placed on a subject, digitized, filtered, transformed and subsetted, and converted to instructions. The results of these instructions are then fed back to the user.

3.1.1 Recording

The recording step is performed in an experimental context, for which an experimental paradigm, including data collection methodology, is necessarily formulated in advance (i.e. go/no-go, oddball, etc.) [13]. In other words, the data produced is expected to be well-defined and well-formatted. EEG electrodes for most research-grade measurement equipment may be repositioned over a great number of possible standard positions on the scalp [89], further contributing to the uniformity of the data.

Data is typically produced as a stream of packets output by the recording equipment, for transport through a standard interface. The resulting data is stored in tabular form [138], although BCI2000, for example, makes the data available in a simple UDP streaming format [3], as well as providing an implementation of SWLDA\(^5\) for initial classification.

The "long" tabular form consists of columns of sampled data, where each row constitutes a single sample. Thus, there are as many rows as there are samples obtained (excluding samples discarded by the experimenter). Columns include the EEG channels, as well as annotations marking events, class labels (in the case of a classification experiments), and experiment state. This form is suited to batch processing and fast queries on the data (that do not require a specific transformation such as CSP\(^6\)).

---

5Stepwise linear discriminant analysis.
6Common spatial pattern [31]
Table 1: "Long" form of epoched EEG data

<table>
<thead>
<tr>
<th>Sample</th>
<th>Trial</th>
<th>Class</th>
<th>Channel</th>
<th>Voltage</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-14.36</td>
<td>-100</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>-15.94</td>
<td>-100</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>-22.85</td>
<td>-100</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>-47.43</td>
<td>-100</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>-65.98</td>
<td>-100</td>
</tr>
</tbody>
</table>

Another form, slightly more amenable to algebraic processing, is a "cubed" format, suitable for epoched data. In this case, the rows of the tabular form are ordered along a third dimension according to experimental trial. This form is much less flexible than the tabular form, and precludes appending many of annotations present in the tabular columns, but they may be stored separately and linked to the structure storing the epoched data.

Figure 4: An example of 'cubed' tensor data (here block triangular data), an expected input structure for a variety of algorithm classes.

3.1.2 Import/Preprocessing

The data as provided by the experimenter and/or recording software must be converted to the most flexible, efficient format possible in order to make data analysis painless. Inefficiencies are felt throughout the entire analysis process.

Unfortunately, there are no approaches at this time that allow the data to be stored to take full advantage of both the tabular and cubed form. As a best compromise between processing speed, memory use, and

---

7 Epoched data, in the context of biosignals, refers to an ordering of the data by channel so that data from repeated trials is arranged so that samples from repeated trials are adjacent.
8 For example, multiple data types are unsupported.
usability, the data imported into rBCI is stored in an R data.table, a high-performance table structure with a powerful and compact syntax for queries [32]. When an operation that requires the cubed form is run on the data (such as a by-trial transform), a query is performed on the data.table such that the data returned is exactly of the structure needed for the operation. This keeps memory use down, and is relatively fast.

On a technical level, there are three main challenges that are typically encountered:

1. Conversion of format
2. Cleanup/removal of erroneous or irrelevant components
3. Storage and recall (availability on disk and in memory)

Challenges 1 and 3 can be handled mainly by automated processing. This is not meant to trivialize the process of conversion; indeed, given the number of formats in use for BCI, a significant amount of programming effort is required to support them all completely (and track changes) [33] [17] [107] [75]. Nevertheless, the goal of bringing the researcher as "close to the data as possible" requires, at a minimum, overcoming these three challenges.

Challenge 2 can be accomplished in conjunction with the user. Indeed, what is erroneous or irrelevant is necessarily contingent on the user’s preference and interest.

As a general principle, data handling or "munging" of this kind should be done in such a way as to reduce the time spent by the user—in some cases it may account for the majority of the time allocated to analysis [134].

1. Example

Consider data split by subject and session into multiple MATLAB files, labeled mechanically. Each file contains a structure eeg with members data, rate, and class, describing the recorded EEG data, the sampling rate, and the class labels respectively. data is a 3-dimensional array (stored as samples x channels x trials), rate is a scalar, and class is a vector of labels with length equal to the number of trials.

In this case, appropriate steps to compactly organize the data might be as follows (required user inputs in bold):

(a) Get the location of the MATLAB files
(b) Get subject and session naming scheme
(c) Import all files into namespace
(d) ’Unfold’ or ‘melt’ [132] the high-dimensional data into a table for each subject and session, recording sample/channel/trial order; affix columns labeling them appropriately (class/subject/session/rate)
(e) Concatenate each table into a single large structure
(f) Save the data as converted to a file as needed

These steps are straightforward, excepting the fourth step, which requires attention to constraints of memory and processing time, as well as care to preserve row uniqueness.

3.1.3 Filtering

"Filtering" in this case refers to simple frequency-selective digital filtering performed in order to remove common sources of noise, and to localize the band of interest. These operations are slightly less agnostic to the data recorded, but still apply across the entire experimental time series.

In the case of EEG, typical frequency noise sources are well-documented [33]. These include:

9i.e., ordering of the data.
• AC line noise (50/60Hz)
• DC offset
• Neuromuscular signals caused by movement
• Anything above the maximum known band of interest for neurosignals, around 40Hz

Thus, the typical approach is to reject known major interference bands, and then follow up by passing only the band(s) of interest.

This can be accomplished quickly, in most cases without any memory overhead. (In general there is a net memory savings, as data may be safely downsampled following a low pass filtering operation.)

1. Example

In the case of P300 analysis, a researcher may wish to apply a low pass filter to remove unneeded high-frequency components (e.g., those greater than 10 Hz). The process is as follows:

(a) Get the filter design parameters
(b) Provide the parameters to a design equation or library to produce a filter object
(c) Apply the filter object to the data, obeying the column grouping of each signal, returning the filtered data

3.1.4 Transformation

Transformations involve large-scale mathematical operations on the data. These operations are intended to extract structure or structural features (such as CSP or PCA), normalize the data according to some metric (i.e., a statistical one, whether parametric or no), or to reduce dimensionality (compressive sensing, among others).

Of all of the steps in EEG-based BCI processing, transformation often consumes the most memory. However, if the features extracted by a transformation are sparsely located in the resulting data, a large amount of data may be discarded at this step, reducing the computational load for the next step.

Transformations finding frequent use in EEG-based BCI include:

• PCA
• CSP: common spatial pattern
• Wavelet

Naturally, the effectiveness of a transform in extracting features of interest depends on the underlying physiological characteristics of the signal source, i.e. the phenomenon under study. This strongly depends on the experimental conditions and interface paradigm—CSP is effective for ERD-based interfaces, as motor imagery is strongly spatially located, for example.

1. Example: CSP for two-option interface

CSP, or common spatial pattern, rests on the assumption that a BCI paradigm’s selections are strongly spatially correlated with the EEG channel (i.e., location). This holds generally for the paradigm of event-related desynchronization, which utilizes motor imagery by the user focused on different areas of the body. Thus, the goal is to design a filter that increases separation between the two channel sets.

10 (i.e., by trial, channel)
Mathematically, this is achieved by jointly diagonalizing the autocorrelation matrices of each trial, resulting in a pair of (unique) eigenvalues and (non-unique) eigenvectors. In order to be useful as a BCI preprocessor, joint diagonalization is performed on the within-class average of autocorrelation matrices.

This boils down to the following steps:

(a) **Subset the data by trial, channel, and/or time interval as directed**
(b) **Compute the autocorrelations of each trial**, then average by class
(c) **Compute the generalized eigenvalues and eigenvectors for the two autocorrelation matrices** (exact solution for the two-class case), sort
(d) **Throw away eigenvalue/vector pairs not needed by the user**
(e) **Transform each trial to obtain either generalized eigenvalues or reduced subspace**

### 3.1.5 Classification/Clustering

**Classification** and **clustering** are included in the domain of machine learning, referring to adaptive algorithms that converge on a specific solution, either in the presence of known outputs (i.e., class labels), or in their absence, defining a **model**. These two approaches are more appropriately known as **supervised** and **unsupervised learning**, but BCIs are mainly concerned with the specific **tasks** of classification and clustering, as they directly relate to the processes of identifying user choice (closing the user interface loop), and locating correlates of user choice, respectively.

Given the great variety of learning algorithms and implementations, it is difficult to directly optimize their processing. However, given the parallel nature of many algorithms, multicore and distributed processing can be remarkably effective at improving their performance across the board. This has the added benefit of allowing classification algorithms to **scale** their performance as datasets increase in size—as many or as few resources may be allocated as needed.

1. **Classification**

   Common classification algorithms used in BCI include:
   - LDA (linear discriminant analysis)
   - SWLDA (stepwise linear discriminant analysis)
   - SDA (shrinkage linear discriminant analysis)
   - SVM (support vector machine; linear and nonlinear kernels)

   Stepwise linear discriminant analysis proceeds through the variables used in initial feature selection, selecting the most correlated predictor first. It adds predictor variables to the discriminant, until the canonical correlation of the model does not increase by a significant amount.

   Shrinkage linear discriminant analysis applies a regularization to the estimate of the covariance matrix, $\Sigma = (1 - \lambda)\Sigma + \lambda I$ (1)

   where $\Sigma$ is the covariance matrix and $\lambda$ is a regularization parameter referred to as shrinkage intensity.

   (The off-diagonal elements of the covariance matrix may also be adjusted by substituting another matrix than the identity matrix for $I$.)

---

11 Many more complex variations on this process exist.
12 As these are real signals, there exist many shortcuts to this computation.
Support vector machines seek a solution to the optimization problem of finding a hyperplane (in the feature space) that maximally separates points drawn from two (or more) unknown distributions. Specifically, SVMs solve the quadratic optimization problem

$$\arg \min_{w,b} \frac{1}{2} \|w\|^2$$

subject to the constraint that for any \(i = 1, \ldots, n\)

$$y_i(w \cdot x_i - b) \geq 1$$

where \(w\) is the vector representing the classifying hyperplane, \(x_i\) is the \(i\)-th point in the feature set, \(b\) is the bias (the displacement of the hyperplane from the origin), and \(y_i\) is the class label of the \(i\)-th point.

Variants of discriminant analysis are favored for their fast computation and convergence. Blankertz makes specific mention of this in [21], in comparing what he considers the two dominant approaches to signal processing toolchains in BCI:

(a) Generic preprocessing and transformation, followed by extremely powerful classification
(b) Complex, application-oriented preprocessing, followed by simple classification

Both of these approaches appear to yield comparable results thus far, suggesting that the choice is best left up to the user\(^{14}\). In either case, a great deal of fine-tuning, dependent on the experimental subject, session, conditions, and data are required. Much of this is accomplished through hyperparameter optimization\(^{14}\).

Hyperparameter optimization is the process of determining optimal values for parameters that in turn govern the convergence of a learning algorithm. These include kernel parameters, trial weights, error costs, and regularization parameters. Since the algorithm itself cannot determine optimal values for these, they must be informed by the data, while avoiding the pitfalls of overfitting. Strategies for this include bootstrap sampling and cross-validation, both of which involve separating the data into multiple sets for training and testing of the learned model.

2. Clustering

Clustering finds use in BCI for feature selection within existing paradigms as well as exploratory research. Measures used vary from simpler ones such as a Davies-Bouldin index\(^{119}\),

$$DB_m = \frac{1}{m} \sum_{i=1}^{m} \max_{j=1, \ldots, m \neq i} \frac{s_i + s_j}{d_{ij}}$$

where the distance measure between classes \(i, j\) and dispersion of each cluster \(s_i\) is combined into a single expression for up to \(m\) classes total, to \(k\)-means clusters optimized by mutual information criteria\(^{66}\).

These clusters can be used within an existing feature space to tighten the BCI’s tuning before the classification step, or to train otherwise redundant classifiers on multiple clusters as a stopgap against EEG variation due to subject and environment state\(^{66}\).

\(^{13}\) In practice, the above equation is rephrased using its geometric dual to avoid computing the dot product between the plane and each of the feature points.

\(^{14}\) CRAN, the R community repository, has a large number of peer-reviewed packages for classification, allowing new users a variety of options for approach 1.
3. Example: SDA with hyperparameter grid search

A simple litmus test of an interface paradigm or processing method can be conducted using LDA with shrinkage (SDA):

(a) Separate data in feature space into training, testing, and validation subsets (by trial) according to given proportions

(b) Establish a subset of hyperparameters (shrinkage for autocorrelation matrix on- and off-diagonal) to search over

(c) Train SDA classifiers on the training subset, test the resulting classifier on the testing set, and record the accuracy. Repeat for each hyperparameter tuple.

(d) Test the best classifier/hyperparameter pair (or all such pairs) on the validation set.
4 Program design

As can be seen from the previous section’s description, the processing of EEG-BCI data begins with operations relatively agnostic to the data, proceeding to those highly dependent on the exact features of the specific data under examination. This informs the design of rBCI.

4.1 Components

rBCI is divided into seven modular components, each of which guides the user through tasks performed on available data. Both the modules themselves and their content are designed for modularity—each module is mostly agnostic to the data it accepts as input, relying on the user to make sensible choices from the data loaded into or created in rBCI environment. Module subcomponents may also easily be added—their interfaces and backend functions can be loaded on the fly during module initialization. This allows rBCI to grow and respond to research and development.

4.1.1 Import Module

The import module allows users to import a variety of data types into the analysis environment. rBCI currently supports two MATLAB structure types, RData, and comma/tab separated values (CSV/TSV).

Users select a data file from a filesystem selector dialog, which can then be inspected in a preview, which allows the user to verify the data has been imported properly, and tweak options if it is not. Depending on the data type, different preview options are available, including:

- Structural: a text-based summary of the file’s structure upon import
- Graphical: a plot of grand means computed from the data (MATLAB Type 2 only)
- Columnar: a truncated table view of the data
- Raw: raw program output of a truncated portion of the data
4.1.2 Exploration Module

The exploration module is typically the first stop after importing a new data file, in order to inspect and partition the data for analysis. Here the data can be examined in a number of summary forms, including histograms, means plots, and column statistics.

Once the user is satisfied with the result, she may either export the data in RData format or bring it into the larger program environment, making it available to other modules for analysis.
Most importantly, here the data can be partitioned randomly or order-contiguously by a user-selected grouping column in order to create sets for testing or validation of subsequent analysis or modeling. This can be done for each dataset (or resulting subset) by selecting the partition number and proportion(s) of the total to divide over.

### 4.1.3 Filter Module

The filter module allows the user to apply simple frequency-based filtering to an existing dataset. In order to keep this module accessible, the interface restricts the user to the following choices:

- Column(s) to which the filter will be applied
- Stop/passband start and end
- Data grouping: i.e., over which groups the filter will be applied (by trial, by channel, etc.)

Once the filter parameters have been set, the user can then apply the filter to selected data, or export the filtered data to the filesystem. The user may also downsample newly filtered data using a similar simple interface in an adjacent tab.
Figure 7: Filter module with available settings displayed.

4.1.4 Transformation/Unsupervised Learning Module

The transform module provides the user with a simple interface through which to execute different unsupervised
learning or transformation algorithms on data. rBCI provides interfaces for principal component analysis, the
common spatial pattern algorithm, and \( k \)-means clustering.
Each interface presents similar options to soften the learning curve:

- **Type**
- **Numerical parameters**
- **Target data columns**
- **Visualization**
- **Data export**

1. **PCA**
   
The PCA module allows the researcher to select the kernel type, its numerical parameters, and the size of the retained subspace. Linear PCA is computed using core R libraries [98], while nonlinear PCA is computed using a general kernel framework [56]. The computed analysis object (containing the subspace data) can be saved as is, applied to a dataset, or plotted for examination.

2. **CSP**
   
The CSP module functions similarly to the PCA module with the exception of the kernel type setting, which is absent. The module uses an implementation developed as part of this thesis [15].

3. **k-means**
   
The k-means module allows the researcher to select from a variety of algorithms [98], the number of clusters to compute (or to infer their number from a heuristic), and the maximum number of times to iterate. The cluster data can be saved as well as plotted.

4.1.5 Classification Module

The classification module allows the user to build, tune, and evaluate supervised learning models on data. rBCI includes linear discriminant analysis with shrinkage (SDA), support vector machines (linear and non-linear kernels), and Naive Bayes classifiers.

Briefly, the Naive Bayes classifier computes the conditional likelihood that a point is a member of a certain class under the assumption that each of the pieces of evidence (features) are completely independent. In other words, it computes the likelihood under independence assumptions using Bayes’ Theorem

\[
\arg \max_{\text{CLASS}} P(\text{CLASS}|e_1, \ldots, e_n) = \frac{\prod_{i=1}^{n} P(e_i|\text{CLASS})P(\text{CLASS})}{P(e_1, \ldots, e_n)}
\]

(5)

where CLASS is a specific class label, \(e_i\) is a piece of evidence (i.e., a feature).

Figure 9: Classification module with naive Bayes classifier settings displayed.

As in the Transformation module, the interface presents the user with a concise set of qualitative options and quantitative parameters to tune. The user then makes the appropriate selections, including model and result output variable names, then begins processing.

The results may be viewed in table format, a summary of the model object itself, or (depending on the model type), graphically. The computed models can be saved and tested on other data sets, summarized, or
plotted in an overview.

1. **SDA** The SDA classifier allows the researcher to specify three different shrinkage coefficients (or leave them at 0 to allow them to be computed heuristically): the shrinkage applied to the correlation matrix (off-diagonals), the shrinkage applied to the variances (diagonals), and the estimated frequencies of the data.

2. **SVM** The SVM classifier presents the user with similar options as the PCA module, namely, a choice of kernels and numeric hyperparameters. Linear SVM is computed using the `LiblineaR` library, while kernel SVM is computed using `kernlab`.

3. **Naive Bayes**
   
   The naive Bayes classifier only has one dial to twist, namely the strength of Laplacian smoothing. The remaining interface elements are the same.

### 4.1.6 Report Review/Generation Module

Whatever steps the researcher takes are tracked internally by `rBCI`. When the researcher opens the report module, they are presented with a list of steps taken.

![Figure 10: Reporting module displaying overview of applied steps.](image)

The user can add each of these steps to a report design, changing their order and title. Once the user is satisfied with the ordering of steps in the report and their composition, they may export the report as HTML and as a further editable Markdown document, using the R literate programming tool `knitr`. This greatly reduces the burden of producing code, documentation, and results in completely separate environments.

### 4.1.7 Tool Designer Module

The tool designer module is a slightly advanced module that provides the user with initial boilerplate code and a separate editing window to design their own tool for analysis using any available R libraries. A file selector is also provided for saving and loading custom tool scripts for later application.
Once the user is satisfied with her design, she may test it on any data in the environment, by selecting it from a drop-down menu. The result, if successful, will also be stored in the environment. The resulting code will also be made available in the reporter GUI, allowing the user to include it in the report as desired.

4.1.8 Performance Options Module

This module sets preferences relating to high performance computing components used in the processing and analysis of data.
The main options are concerned with parallelism. In the case of local parallelism, the user need only configure the number of cores she would like to allow processing to use. In the case of distributed parallelism, the user needs to configure the network connection (typically SSH) and other necessary information to communicate with the distributed machines. However, this only need be done once per session, at which point the settings will be saved.

4.2 Use Cases

4.2.1 Classification of epoched P300 data with PCA, SDA, and SVM

This case study was published in the proceedings of the Asia-Pacific Signal and Information Processing Society in 2014 [8]. We summarize its results here and show how the program designed here can be used to replicate the EEG data analysis steps given in that study.

The experiment and processing steps for the case study in question were as follows:

A typical BCI research task is the validation of a proposed interface (i.e., the actual physical design) for possible clinical use. This involves the analysis of experimental data collected using the interface to determine the accuracy obtainable from this interface. In this case, the physical components are the object of study, while the algorithms for processing and classification are chosen from the set of those used in comparable published literature.

1. Experiment
(a) instruct subjects  
(b) present stimuli  
(c) get data  
(d) preprocess

2. Data Processing

(a) import  
(b) filter  
(c) transform  
(d) classify

This case study makes use of P300 experiment data collected from a novel tactile BCI experiment. Notably, stimulus was presented using a contactless ultrasound device (airborne ultrasonic tactile display, or AUTD). Data was sampled at 512Hz, with the stimulus length and interstimulus interval both 400ms. Initial processing included a bandstop filter from 48-52Hz for power line interference, and a bandpass filter from 0.1-60Hz. Thirteen male subjects participated in three experiment sessions, which each contained a randomly ordered set of 90 targets and 450 non-targets. The resulting time series was epoched by channel into separate trials, over the interval -100 to 900 milliseconds (pre- and post-stimulus, with overlapping samples duplicated).

In the following subsections we describe how the data processing for the above case study can be accomplished using high performance computing techniques with rBCI without requiring specialized programming knowledge.

1. Import

We provide sample data here in a format we have termed "MATLAB Type 2", as it is the second arrangement of MATLAB data encountered in our research. This structure is arranged as follows (inner nodes are contained in outer nodes):

(a) Binary MATLAB data file
   i. MATLAB structure eeg
      A. Epoched EEG data in 3-dimensional array: samples x channels x trials  
      B. Class label binary vector in 1-dimensional array: 1 x trials  
      C. Sampling rate: scalar

The researcher begins by selecting the file from her filesystem through a modal selector. The order of the structure components may vary, so the next step is to select the EEG and Class label indices within the structure. At this time, this is all that is needed to import the data, so the researcher can now select Preview to attempt an import and make sure it is successful by displaying a diagnostic output. The current choices available for this format are Structural, which displays the MATLAB data structure as imported (the equivalent form in R is a list), or Graphical, which computes the grand trial means separated by channel and class with minimal overhead.
Once the researcher has confirmed that the data can be imported properly, clicking **Import into interface** will bring the data into the environment and GUI, making it available to other modules.

Once this is done, the researcher can move straight on to filter the data.

2. Filtering

The next step is to remove high frequencies from the data, specifically, frequencies above 10Hz. The **Simple Filtering** module allows a researcher to automatically design a Butterworth filter of appropriate order, given the normalized frequencies and filter type. The module also requires the researcher to specify the grouping columns, which are typically the epoch and channel columns.
Figure 14: Filtering to pass only frequencies below 10 Hz, using a designed Butterworth filter.

Once the parameters are set, the researcher can apply the filter to the data by selecting **Apply Filter to Data**, which displays a magnitude plot of the filter and applies it to the data, grouped by the selected columns. If the researcher is satisfied with the result, she can then select **Save Filtered Data** to bring the result into the program environment, just as was done with the imported data.

In order to reduce the amount of data needed for processing, the researcher may also choose to downsample the resulting data by a fixed factor. In this case, we elect to downsample to 2Hz above the Nyquist rate to leave a margin of error.
3. Set Separation

To ensure the result is generalizable, it is also necessary to split the data into a minimum of two sets: training and testing. The researcher can do so by selecting Explore Data from the main menu, and using the Partition module.

Here the researcher can set the partition division for 2 sets (a number between zero and one) to an appropriate value (in this case, we select 0.6). This generates a random vector of indices sampled without replacement from the partition group.
4. Transformation

Now that the researcher has filtered and separated data to work with, the next step is to extract relevant features. EEG signals are typically highly correlated, requiring some method of orthogonalization to extract signals of interest. Here we make use of principal component analysis to do so.
Figure 17: PCA module configured to compute the principal components from a selected data set with specified groups.

The procedure is similar to the filtering task: the researcher selects the data set, PCA kernel type (here Linear) and the number of features (principal components) to return from the computation. Once the researcher is satisfied with her choices, computation is begun by selecting Compute PCA. This computes the PCA from the data, bringing the PCA model into memory.

At this point, the researcher can explore the model by selecting Plot Eigenvalues (i.e., the variance along each principal component dimension), or transform the data with Transform Data Set to find the principal components for each Trial, using the model computed in the previous step. Note that the model computed from the first partition is used to transform the second, whose data is held for testing.
Now that we have features, the next step is to test them with a classifying algorithm.

5. Classification

In the classification module, the researcher can select the PCA-computed features from the training set, train a model on them, and then verify them on the testing set. In this case, we will use linear SVM from the LiblineaR library [51].

The process is similar to applying PCA: the researcher selects Kernel Type from a list, and sets Numerical Parameters based on the chosen type. Once the researcher is satisfied with the configured parameters, the model can be trained using the Train button, which will train an SVM and output a model to the data set list. Likewise, a trained model, once tested, can be quickly examined here for accuracy.
Figure 19: Reviewing the generated SVM model.

Once the model has been tested, the results can be printed to the neighboring pane by selecting **Print Table**. The model itself can also be printed with **Print Model**.
As mentioned previously, a minimum classification accuracy for BCIs is 70% \cite{112}. If the printed model accuracy is greater, then we can conclude tentatively that the proposed interface is viable.

4.2.2 Exploring behavioral correlates with k-means

As another example, we consider the following problem: What if a researcher wants to investigate a body of experimental data for possible correlates of intent to discover a new BCI paradigm? Clearly, a supervised learning model is not general enough to make this task efficient. However, clustering might provide enough evidence for the researcher to formulate a testable hypothesis (provided those hypotheses are then retested on different data).

Assume the researcher has epoched multichannel EEG data which may contain evidence of user intent, but cannot be sure how it is manifested or spatiotemporally localized. In this case, there is no **Target** column, and supervised learning/classification cannot be carried out. Where steps are repeated from the **Classification use case**, we abbreviate them here.

1. **Import**

   Here we demonstrate loading from a different file format, namely RData, a format native to R.
In this case, the data is tabular in nature (with further dimensions instead marked by extra column permutations). The RData import module allows the researcher to preview a set number of rows, and mark the desired columns for import. Import is otherwise the same as in the previous section.

2. Unsupervised Learning

Since we do not know the frequency band of interest, it may be premature to filter the signal. Instead, we move straight to the Transform/Clustering module to apply the k-means algorithm. From a user perspective, the process is similar to the PCA option: the researcher selects the algorithm of choice from a list and the numerical parameters that apply.

In this case, the researcher selects the number of cluster (0 allows the algorithm to make a guess at the appropriate number) and the number of iterations to compute. Once satisfied with the configuration and Output.Variable for the model, we select Cluster Data. Once the model calculation is complete, the result can be plotted with Plot Clustered Data.
3. Transformation

The same calculation can be performed on transformed data. For example, the researcher can compute the principal components of the data on the PCA tab as in the previous study, then return to the k-means tab to recompute clusters on the resulting data.

PCA can be carried out in the same fashion as in the previous case, as illustrated in the below plot.
This data can be immediately clustered in the k-means tab. The following plot illustrates the result of clustering with two centers and the first three principal components:

![Figure 24: Clustering on the first three principal components of test data](image)

In this case, the experimental data is not amenable to clustering without further knowledge of features of interest, such as frequency or time.

### 4.3 Technical Overview

From a technical perspective, rBCI is made up of two major subsystems—the GTK frontend GUI, and the backend processing API. Both are written entirely in R\(^{16}\), and are written in a mixed functional/imperative style.

#### 4.3.1 Frontend GUI

The frontend GUI is written using gWidgets\[^{115}\], a framework for instrumenting common GUI elements across toolkits (e.g., Tcl, Gtk2, Java) using a platform-independent syntax. As a compromise between richness of features and compatibility across operating systems, the GTK2 (GIMP Toolkit 2) was chosen as the underlying GUI toolkit, using the library RGtk2\[^{63}\].

Each task module is implemented as a base gWidgets script that initializes the task window and common components\[^{17}\]. Once loaded, the base script calls multiple submodule scripts that contain the actual options, output views, and calls to backend functions.

GUI actions that change the state of the environment (either directly or through a call to a backend function) are recorded by the reporter module by means of a call to the reporter backend. The GUI handler partially deparses the actual backend function call until it is portable enough to run without reference to any part of the program state except for imported/generated datasets, then calls a backend function to add the call and a short summary description to the stored step list.

On a more technical level, the code generation feature provided to store analysis steps as calls identical to those run by the triggered handler is a form of metaprogramming\[^{26}\], a procedure in which a program is

\(^{16}\)Excepting included libraries.

\(^{17}\)Such as data set selectors.
used to generate code that it itself runs. This is aided by the homoiconicity of R: language calls themselves are first-class data types, and as such may be manipulated using the same operators as any other object.

4.3.2 Backend API

The backend consists of two main components: an initialization script that sets up fundamental storage structures, and a collection of functions that carry out the actual tasks of analysis.

The initialization script loads necessary packages (installing as needed), sets up the main rBCI environment and defines lists within it for storing imported and generated datasets, options for parallelism, and a list of function calls that store analysis steps as called. In other words, the rBCI environment structure stores the state of the program.

Backend functions consist of two types: utility functions for minor work (such as curried functions for binding matrices along specific dimensions), and functions that constitute semantic analysis steps (e.g., `train_svm_model()`). These functions are defined within the global scope, making the backend a public API, which enables more advanced users to build on existing implementation and extend the program.

Backend functions are normally called by GUI handlers, which pass user-selected elements to the function along with relevant environment/state as needed. As described above, calls that create data as part of the analysis are passed to the reporter backend for later export.

Once configured and invoked, the reporter backend exports the analysis environment, backend helper functions, and program calls dynamically generated by the researcher’s actions within each GUI module, to a directory chosen by the researcher. Program calls are parsed and partially dereferenced to a minimum required for portability, then converted to strings to be placed in separate evaluation sections of the human-readable R Markdown document. This document is then parsed for code and Markdown text by `knitr`, which passes the exported code to the calling R session for evaluation, returning a formatted code block and the results of the call in its place. The Markdown text is then formatted as HTML, for which Markdown provides a feature subset.

---

18 A special list structure that affects evaluation scope.
19 This is kept to a bare minimum.
5 Optimizations

Here we discuss performance optimizations embedded in rBCI. They can be divided roughly into two types: optimizations for memory use, and optimizations for processing speed, specifically a variety of types of parallelism. Example processes are benchmarked, for which specific code and published results are publicly available [2] [3].

5.1 Memory Optimizations

5.1.1 Strict call-by-reference

In R (as in MATLAB, for example), values are passed between calling functions by reference, except when modified. When the calling function modifies them, a complete copy is made. This is copy on edit, a form of lazy evaluation [25]. This strategy saves memory if a function is expected to return only values derived from its inputs, leaving the original data intact. However, in BCI, the original data is expected to go through a number of sequential processing steps before useful results are produced, making this an extremely inefficient and wasteful approach—multiple copies of the data are spun out for every step that may modify the data, from filtering and downsampling to transformation. This can quickly create an overhead four or more times the size of the initial dataset, which is cost-prohibitive for current (and continually increasing) data set sizes and machines 21.

An effective solution is to enforce pass-by-reference in all cases. In all processing steps, we make use of an extremely efficient structure called data.table, for which all transformations on the data itself are strictly call-by-reference [22]. It has a concise and readable syntax that works as follows:

```r
query <- some.table[where, select|update, by]
```

where select|update may be any function that can be evaluated within the context of the table. The by statement determines how data is grouped when selected: for example, when filtering an epoched dataset, one need only group by = c("Trial","Channel") 22.

The where statement can be used to accelerate processing dramatically when working with specific queries. data.table performs binary searches (O(log n) complexity) on queries using the tree’s key(s), using a radix sort. In other words, a query performed in this manner,

```r
setkey(some.table,Trial,Channel)
spectrum.table <- some.table[J(1:100,c(5,6)),
                        get.spectrum(Voltage)]
```

will quickly return a table containing the spectral content of channels 5 and 6, from trials 1 to 100. The setkey() function sorts the rows by reference so that values with neighboring keys are contiguous in memory. The J() function simply joins two queries so that data.table knows to perform a binary search and not a vector scan.

---

20 As we observed firsthand in working with data in both MATLAB and R.
21 The main data set treated here, for example, is roughly 10GB, processed on a machine with 32GB of memory.
22 These commands can also be chained by appending another bracket to the same statement; e.g. `dt[call1][call2]`.
5.2 Processing Optimizations

5.2.1 Explicit parallelism

The main optimization performed to increase processing speed is explicit parallelism. Explicit parallelism refers to the programming style of explicitly naming the processing elements that are meant to run in parallel, affording the programmer a measure of control at the cost of requiring a degree of forethought. rBCI employs explicit parallelism behind the user interface, exposing only a set of parallel backends that the user can choose from. This modular approach allows the user to drastically scale up processing to grid-level parallelism while preserving the same workflow.

1. Frontends

The main parallel frontend used is \texttt{foreach}, an R library that provides an iterative framework for job dispatching. The \texttt{foreach} syntax is formulated in a natural manner:

```r
output.var <- foreach(iterated.var = iter(input.var),
            .combine=cbind) %dopar% {
    function.of(iterated.var)
}
```

The \texttt{foreach} construct as given above iterates over the \texttt{input.var} provided, dispatching the block after \texttt{%dopar%} to the parallel backend provided, namely some function of the \texttt{input.var} wrapped in an iterator. The value of the last statement valued inside the dispatched block is returned to the dispatcher and combined or aggregated using the function passed to the .\texttt{combine} argument. This allows not just simple iterative processing to be easily rewritten for parallelism, but also makes any computation that can be phrased as a map and reduce extremely concise. For example, computing the average spectral content in parallel for all trials for which a p300 is expected to appear could be concisely formulated as:

```r
setkey(eeg.table,Class)
avg.class1.spectrum <-
    foreach(this.class = eeg.table[,which(Class==1)],
            .combine=mean) %dopar% {
        get.spectrum(eeg.table[this.class,Voltage])
    }
```

This would be an effective approach when there are an extremely large number of trials for which to compute the spectrum. Additionally, steps requiring multiple levels of iteration can be combined by wrapping the inner \texttt{foreach} expression in another, with \texttt{:%} as the keyword instead of \texttt{%dopar%}. (See the appendix for further examples.)

2. Backends

As explained above, \texttt{foreach} functions with a variety of parallel backends in a modular fashion. This means that the same code written once can be used in a variety of high performance computing environments, both local and distributed. This setting is changed by ‘registering’ a backend with \texttt{foreach}, which typically requires changing only a short snippet of code separate to the actual processing code. Below we give a few examples.
Local parallelism: doMC, doParallel

Local parallelism consists of distributing parallelized tasks to subprocesses (through process forking, sockets, or other operating system features) on the local machine. This ultimately results in tasks being dispatched to multiple cores (or multiple threads if the CPU is also hyperthreaded). Here we introduce two parallel backends for local parallelism.

i. doMC

doMC is the most common backend used for local parallel processing. It spawns forked R processes equal to the number of cores registered, each sharing the memory of the first to avoid excessive overhead\(^{23}\).

The registration process looks like the following:

```r
library(doMC)
registerDoMC(cores=4)
```

Since doMC forks the current R process, it inherits all of the variables in the current R environment, making it easier to use than the others. However, it can only address local cores.

ii. doParallel

doMC is a newer and more user-friendly local backend, but the cost is paid in manageability and compatibility. Each time a parallel task is dispatched, doMC creates additional forked processes, accumulating overhead\(^{24}\). Moreover, Windows does not have the ability to fork processes, making doMC unusable on these machines.

doParallel is an older but more flexible backend supporting small-scale and local clusters\(^3\). Local clusters must be explicitly specified and managed with doParallel (more specifically, the R core package parallel\(^9\)), but this ensures that whatever process overhead they incur is strictly bounded.

This is done by defining a Cluster object:

```r
require(doParallel)
cl <- makeCluster(3) # 3 local workers (cores)
registerDoParallel(cl)
```

Processing is otherwise the same (due to modularity), but the created Cluster should be stopped after processing is done:

```r
stopCluster(cl)
```

Distributed parallelism: doSNOW, doMPI, doRedis

There are a number of backends developed for cluster computing which utilize well-known cluster computing standards. We outline them briefly here and give examples of code needed to make them available to foreach\(^{24}\).

---

\(^{23}\) This requires a small degree of care, as processes can interfere with each other if poorly programmed.

\(^{24}\) Setup and administration of these cluster types are out of the scope of this document and depend on available facilities, although tutorials are readily available for Amazon AWS\(^{11}\), a local network reachable through SSH\(^{81}\), and MPI with Amazon EC2\(^{67}\).
i. doSNOW
SNOW stands for 'Simple Network of Workstations', a small-scale distributed computing system for R [101]. As its name implies, it is aimed at operating clusters of workstations, i.e., machines already provisioned for use at an institution or office. SNOW provides a unified interface to three different process-to-process communications: sockets, PVM (parallel virtual machine) [44], and MPI (message passing interface) [46]. SNOW clusters can be reached through doSNOW [10], a backend for foreach, as follows:

```r
cl <- makeCluster(2,
  type="SOCK")
registerDoSNOW(cl) # registers cluster with foreach
```

Similarly to doParallel, SNOW clusters must be stopped after a processing session is complete, to deallocate resources.

```r
stopCluster(cl)
```

ii. doMPI
The doMPI package provides direct access to MPI clusters without the SNOW wrapper. This is useful in cases where connecting to an MPI cluster requires more advanced configuration, as in the case of Amazon EC2 [67]. Once the cluster is set up, registering it with foreach is done similarly as with doSNOW:

```r
cl <- startMPIcluster(count = 3)
registerDoMPI(cl)
stopCluster(cl)
```

doMPI automatically loads each core registered to 100%, making the `stopCluster()` command critical [91].

iii. doRedis
Redis is a performant distributed key-value store that can be used as a task queue, making it an appropriate backend for managing parallel jobs [103]. It can be connected to foreach through the doRedis package [64]. Amazon makes a dedicated Redis service available called ElastiCache, enabling large-scale processing with the same code [42]. Redis clusters are extremely easy to connect:

```r
library('doRedis')
redisWorker('thisjobqueue')
```
# run on master R machine

library('doRedis')

registerDoRedis('thisjobqueue')
6 Benchmark Results

Each benchmark was run on two datasets: one from a published P300 experiment provided by the BCI-lab-group at Tsukuba University, and the P300 speller dataset (IIb) from the 2003 Berlin BCI competition.

6.1 Memory

Each benchmark compares two approaches: the data.table approach explained in the previous section, and a lazy-evaluation approach in line with standard R approaches. The criterium we measure is memory released by the function call (i.e., overhead). Benchmarks were computed with the help of lineprof, a memory profiler.

For clarity, we omit the benchmarking code here and give only the actual processing expressions as called (the full code is available in the appendix).

1. Grand means
   A quick way to verify that one is working with valid data for BCI is by computing the means over all trials for every channel. This involves aggregating a significant amount of data, which makes memory savings valuable.

   The two queries tested were:

   ```r
   test.table[, mean(Voltage), by = c("Sample", "Channel", "Class")]
   ddply(test.table, .(Sample, Channel, Class), summarize, 
   mean = mean(Voltage))
   ```

   The two queries were tested on the first 100 trials of the available data.

2. Mean spectral content by channel
   Another memory-costly process is computing the spectral content of a multivariate time series. Since we already benchmarked the mean separately, we precompute the mean in advance and compare only the spectral transformation.

   The function used to calculate the spectrum of a signal is also included. It makes use of an R signal processing library.

   ```r
   get.spectrum <- function(x) {
     spec.obj <- spectrum(x, method = "ar", plot = FALSE)
     outlist <- list()
     outlist <- 20 * log10(spec.obj$spec)
     return(outlist)
   }
   erp.trial.mean[, get.spectrum(V1), by = c("Channel", "Class")]
   ```

25 Specifically, a library called plyr.

51
3. Windowed average

Since many BCI paradigms depend on time-localized features, windowed averaging is also a useful feature. Since windowed averaging is a rolling function of a subset of rows, we also take advantage of a fast C++ library for this task [125] [35].

```r
ddply(erp.trial.mean, .(Channel,Class), summarize,
     spec=get.spectrum(V1))
```

4. Filtering

As discussed earlier, filtering is a critical operation to optimize for memory use.

```r
test.table[,roll_mean(Voltage, n = 10),
    by=c("Subject","Trial","Channel")]
ddpoly(test.table, .(Subject, Trial, Channel), summarize,
    wavg = roll_mean(Voltage, n=10))
```

5. Signal scaling

This is the only case where the naive implementation uses no more overhead than the strict call-by-reference implementation. This is mainly because a scaling operator (i.e., one that centers a set of numbers around mean zero and variance 1) is mapped one-to-one to each element of the table—there is no overhead in either case.

```r
erp[,Voltage:= scale(Voltage)]
```

6.2 Results

Results are also given in terms of the percentage of the data size. All benchmarks were performed on the following hardware:

<table>
<thead>
<tr>
<th>Table 2: Benchmark system specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
</tr>
<tr>
<td>Memory</td>
</tr>
<tr>
<td>OS</td>
</tr>
<tr>
<td>R</td>
</tr>
</tbody>
</table>
Table 3: Memory overhead by task (AUTD experiment, call-by-reference)

<table>
<thead>
<tr>
<th>Task</th>
<th>Memory released, reference</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grand means</td>
<td>0 Mb</td>
<td>0%</td>
</tr>
<tr>
<td>Spectrum</td>
<td>0 Mb</td>
<td>0%</td>
</tr>
<tr>
<td>Windowed average</td>
<td>0 Mb</td>
<td>0%</td>
</tr>
<tr>
<td>Filtering</td>
<td>948.44 Mb</td>
<td>214%</td>
</tr>
<tr>
<td>Signal scaling</td>
<td>0 Mb</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 4: Memory overhead by task (AUTD experiment, lazy evaluation)

<table>
<thead>
<tr>
<th>Task</th>
<th>Memory released, lazy</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grand means</td>
<td>2393.2 Mb</td>
<td>541%</td>
</tr>
<tr>
<td>Spectrum</td>
<td>0 Mb</td>
<td>0%</td>
</tr>
<tr>
<td>Windowed average</td>
<td>2993.8 Mb</td>
<td>677%</td>
</tr>
<tr>
<td>Filtering</td>
<td>2027.18 Mb</td>
<td>458%</td>
</tr>
<tr>
<td>Signal scaling</td>
<td>0 Mb</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 5: Memory overhead by task (BCI competition, call-by-reference)

<table>
<thead>
<tr>
<th>Task</th>
<th>Memory released, reference</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grand means</td>
<td>0 Mb</td>
<td>0%</td>
</tr>
<tr>
<td>Spectrum</td>
<td>0 Mb</td>
<td>0%</td>
</tr>
<tr>
<td>Windowed average</td>
<td>0 Mb</td>
<td>0%</td>
</tr>
<tr>
<td>Filtering</td>
<td>401.84 Mb</td>
<td>32.1%</td>
</tr>
<tr>
<td>Signal scaling</td>
<td>3796.1 Mb</td>
<td>303%</td>
</tr>
</tbody>
</table>

Table 6: Memory overhead by task (BCI competition, lazy evaluation)

<table>
<thead>
<tr>
<th>Task</th>
<th>Memory released, lazy</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grand means</td>
<td>9509.02 Mb</td>
<td>759%</td>
</tr>
<tr>
<td>Spectrum</td>
<td>0 Mb</td>
<td>0%</td>
</tr>
<tr>
<td>Windowed average</td>
<td>104.94 Mb</td>
<td>8.38%</td>
</tr>
<tr>
<td>Filtering</td>
<td>7465.34 Mb</td>
<td>596%</td>
</tr>
<tr>
<td>Signal scaling</td>
<td>7243.65 Mb</td>
<td>578%</td>
</tr>
</tbody>
</table>

As the results indicate, strictly enforcing call-by-reference results in little to no overhead in the majority of the tasks. The importance of choosing such an approach is highlighted by the fact that the percentage overhead for the naive approach is typically well over 100%, in one case running to nearly 800% of the initial data set size; since the amount of available memory must be greater than the maximum required,

6.3 Processing

Parallel processing benchmarks were performed on the same hardware as that used for memory benchmarks for consistency (we reproduce the table below). Thus, only local parallel backends were tested, as the degree of variation among distributed installations is too great to provide a ballpark measurement for.
Table 7: Benchmark system specifications

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon E3-1225 V2 @ 3.20GHz (4 cores)</td>
</tr>
<tr>
<td>Memory</td>
<td>32GiB DIMM DDR3 Synchronous @ 1333MHz</td>
</tr>
<tr>
<td>OS</td>
<td>Debian Linux (kernel version 3.12-1-amd64)</td>
</tr>
<tr>
<td>R</td>
<td>version 3.1.0</td>
</tr>
</tbody>
</table>

Benchmarks were performed on the following tasks:

- Filtering
- Hyperparameter optimization (SDA and SVM, grid search)

We give a short description of each task, including the benchmarked code. Results and analysis are given at the end of the section.

1. Filtering

Parallel and serial processing were compared by filtering the two datasets with a 10Hz Butterworth filter. This process was repeated 20 times with 1 core, 2 cores, and 3 cores and microbenchmarked \[78\]. Example code follows:

```r
filter.seri <-
expression(erp[1:100000000,
    filtfilt(low.pass, # a pregenerated filter object
              by=c("Trial","Channel"))))

filter.seri <-
expression(erp[1:100000000,
    foreach(this.trial=iter(erp[, unique(Trial)]),
       combine=c) %dopar% {
    filtfilt(low.pass,erp[J(this.trial),
                           Voltage])})
```

2. SDA Hyperparameter Grid Search

For each learner, we measured the learner accuracy over the chosen hyperparameter subsets. For shrinkage SDA, we searched over subsets of shrinkage intensity for the correlation matrix lambda and shrinkage intensity for the variances lambda.var. For linear SVM, we searched over subsets of the cost.

```r
foreach(this.lambda=iter(subset$lambda),
       .combine=cbind) %:%
foreach(this.lambda.var=iter(subset$lambda.var),
       .combine=rbind) %dopar% {
    sda.model <-
    sda(lambda.freqs = 1,
```
lambda=this.lambda,
lambda.var=this.lambda.var,
Xtrain=as.matrix(training.pca[,1:3,
        with=FALSE]),
L=training.pca[,as.factor(Class)])
sda.pred <-
predict(sda.model,
        as.matrix(testing.pca[,1:3,
        with=FALSE]))
acc <- sum(sda.pred$class !=
        testing.pca[,Class]) /
        testing.pca[,length(Class)]
if (acc < 0.5) { acc <- 1 - acc}

foreach(this.cost=iter(costs.subset),
        .combine=rbind) %dopar% {
    svm.model <-
        LiblineaR(wi = class.weights,
        cost = this.cost,
        type=3, # cross=3,
        data = train.pca[,1:3,
        with=FALSE],
        labels = train.pca[,Class])

    svm.pred <- predict(svm.model,
        newx=test.pca[,1:3,
        with=FALSE])

    acc <- sum(svm.pred$predictions !=
        test.pca[,Class])/
        length(svm.pred$predictions)
}

6.4 Results

Results for parallelized processes are examined within the context of scalability. Parallelization has been mostly ignored in the BCI literature, and the most popular toolbox, EEGLAB, does not explicitly provide for parallelization in its code [28]. Given the increase in EEG data set size and future proliferation thereof (such as that
proposed by the Neural Engineering Data Consortium [38]), scalability is critical, as indicated by Gustafson’s law [412].

![Speedup by non-parallelizable fraction](image)

**Figure 25:** Illustration of expected speedup with increasing cores for different values of $\alpha$ according to Gustafson’s law.

Gustafson’s law states that given the "scaled speedup" or ratio of the serial process runtime to the parallel runtime $S(P)$, where $P$ is an integer number of processors, the following relation holds:

$$S(P) = P - \alpha(P - 1)$$  \hspace{1cm} (6)

Here, $\alpha$ represents the serial fraction of the process, that is, the fraction that cannot be parallelized. In other words, the law describes the scalability of a calculation—it answers the question, "How much more can we achieve by throwing more power at this process?"

For each benchmark we estimate $\alpha$ from the median runtime following Gustafson’s law.

### 6.4.1 Filtering

Summary runtimes for the filtering process are given below, in table and boxplot form.

<table>
<thead>
<tr>
<th>Type</th>
<th>Min</th>
<th>Median</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>59.82971</td>
<td>60.0070</td>
<td>60.53914</td>
</tr>
<tr>
<td>parallel, 2 cores</td>
<td>56.01123</td>
<td>57.2282</td>
<td>57.77827</td>
</tr>
<tr>
<td>parallel, 3 cores</td>
<td>44.92327</td>
<td>45.61265</td>
<td>46.04711</td>
</tr>
<tr>
<td>parallel, 4 cores</td>
<td>39.16282</td>
<td>41.14741</td>
<td>41.58670</td>
</tr>
</tbody>
</table>
Table 9: Filtering runtimes (s) with increasing parallelism (BCI competition)

<table>
<thead>
<tr>
<th>Type</th>
<th>Min</th>
<th>Median</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>58.21709</td>
<td>58.63809</td>
<td>58.81008</td>
</tr>
<tr>
<td>parallel, 2 cores</td>
<td>18.13518</td>
<td>18.39468</td>
<td>19.02425</td>
</tr>
<tr>
<td>parallel, 3 cores</td>
<td>15.01251</td>
<td>15.46725</td>
<td>17.02028</td>
</tr>
<tr>
<td>parallel, 4 cores</td>
<td>13.60766</td>
<td>14.12618</td>
<td>58.91204</td>
</tr>
</tbody>
</table>

Filter runtime with increasing parallelism

Figure 26: Box and whisker plot of filter runtime with increasing parallelism (AUTD experiment).
From the runtime results, we may roughly estimate the non-parallelizable fraction $\alpha$ (the portion that is responsible for the nonlinear decrease in time) of the filtering process using Gustafson’s law [48].

Table 10: Estimated parallelizable fraction of filter process (AUTD data)

<table>
<thead>
<tr>
<th>$S(P)$</th>
<th>$P$</th>
<th>$\alpha$</th>
<th>Parallel fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.049</td>
<td>2</td>
<td>0.951</td>
<td>4.9%</td>
</tr>
<tr>
<td>1.316</td>
<td>3</td>
<td>0.842</td>
<td>15.8%</td>
</tr>
<tr>
<td>1.458</td>
<td>4</td>
<td>0.847</td>
<td>15.3%</td>
</tr>
</tbody>
</table>

Table 11: Estimated parallelizable fraction of filter process (BCI competition)

<table>
<thead>
<tr>
<th>$S(P)$</th>
<th>$P$</th>
<th>$\alpha$</th>
<th>Parallel fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.187</td>
<td>2</td>
<td>0.0</td>
<td>100.0%</td>
</tr>
<tr>
<td>3.791</td>
<td>3</td>
<td>0.396</td>
<td>60.4%</td>
</tr>
<tr>
<td>4.151</td>
<td>4</td>
<td>0.151</td>
<td>84.9%</td>
</tr>
</tbody>
</table>

Thus we can expect that even under pessimistic conditions, 15% of the filtering process is parallelizable on the testing machine (and its corresponding overhead). Under this estimate, the same 100 million data points from the first set can therefore be filtered in roughly 10 seconds on a relatively small 32-core cluster.

### 6.4.2 SDA

Summary runtimes for the SDA hyperparameter grid search are given below in table and boxplot form.
Figure 28: Box and whisker plot of SDA hyperparameter grid search runtime with increasing parallelism (AUTD experiment).
Figure 29: Box and whisker plot of SDA hyperparameter grid search runtime with increasing parallelism (BCI competition).

Table 12: SDA runtimes (s) with increasing parallelism (AUTD experiment)

<table>
<thead>
<tr>
<th>Type</th>
<th>Min</th>
<th>Median</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>210.72354</td>
<td>212.20380</td>
<td>212.40739</td>
</tr>
<tr>
<td>parallel, 2 cores</td>
<td>100.76444</td>
<td>111.45896</td>
<td>111.86055</td>
</tr>
<tr>
<td>parallel, 3 cores</td>
<td>81.09276</td>
<td>83.45086</td>
<td>85.63844</td>
</tr>
<tr>
<td>parallel, 4 cores</td>
<td>66.55676</td>
<td>67.30343</td>
<td>68.41378</td>
</tr>
</tbody>
</table>

Table 13: SDA runtimes (s) with increasing parallelism (BCI competition)

<table>
<thead>
<tr>
<th>Type</th>
<th>Min</th>
<th>Median</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>381.4667</td>
<td>387.1435</td>
<td>212.4739</td>
</tr>
<tr>
<td>parallel, 2 cores</td>
<td>204.1311</td>
<td>205.6445</td>
<td>111.8055</td>
</tr>
<tr>
<td>parallel, 3 cores</td>
<td>150.1122</td>
<td>151.9375</td>
<td>85.63844</td>
</tr>
<tr>
<td>parallel, 4 cores</td>
<td>114.6514</td>
<td>125.5533</td>
<td>68.41378</td>
</tr>
</tbody>
</table>

Table 14: Estimated parallelizable fraction of SDA hyperparameter search (AUTD experiment)

<table>
<thead>
<tr>
<th>S(P)</th>
<th>P</th>
<th>α</th>
<th>parallel fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.904</td>
<td>2</td>
<td>0.096</td>
<td>90.4%</td>
</tr>
<tr>
<td>2.543</td>
<td>3</td>
<td>0.229</td>
<td>77.1%</td>
</tr>
<tr>
<td>3.153</td>
<td>4</td>
<td>0.282</td>
<td>71.8%</td>
</tr>
</tbody>
</table>
Table 15: Estimated parallelizable fraction of SDA hyperparameter search (BCI competition)

<table>
<thead>
<tr>
<th>S(P)</th>
<th>P</th>
<th>α</th>
<th>parallel fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.883</td>
<td>2</td>
<td>0.117</td>
<td>88.3%</td>
</tr>
<tr>
<td>2.548</td>
<td>3</td>
<td>0.223</td>
<td>77.4%</td>
</tr>
<tr>
<td>3.083</td>
<td>4</td>
<td>0.306</td>
<td>69.4%</td>
</tr>
</tbody>
</table>

As seen above, the parallelizable fraction is quite high, at least 65%. This is expected, since the parameter search process is entirely iterative.

6.4.3 SVM

Summary runtimes for the SVM hyperparameter grid search task are given below in table and boxplot form.

SVM grid search runtime with increasing parallelism

Figure 30: Box and whisker plot of linear SVM hyperparameter grid search runtime with increasing parallelism (AUTD experiment).
Figure 31: Box and whisker plot of linear SVM hyperparameter grid search runtime with increasing parallelism (BCI competition).

Table 16: Linear SVM runtimes with increasing parallelism (AUTD experiment)

<table>
<thead>
<tr>
<th>Type</th>
<th>min</th>
<th>median</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>5182.834</td>
<td>5288.831</td>
<td>5315.910</td>
</tr>
<tr>
<td>parallel, 2 cores</td>
<td>2939.684</td>
<td>2965.018</td>
<td>2989.479</td>
</tr>
<tr>
<td>parallel, 3 cores</td>
<td>2146.640</td>
<td>2172.414</td>
<td>2193.440</td>
</tr>
<tr>
<td>parallel, 4 cores</td>
<td>1776.822</td>
<td>1790.169</td>
<td>1799.507</td>
</tr>
</tbody>
</table>

Table 17: Linear SVM runtimes with increasing parallelism (BCI competition)

<table>
<thead>
<tr>
<th>Type</th>
<th>min</th>
<th>median</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>16841.19</td>
<td>16899.09</td>
<td>16968.34</td>
</tr>
<tr>
<td>parallel, 2 cores</td>
<td>8777.521</td>
<td>8827.217</td>
<td>8857.487</td>
</tr>
<tr>
<td>parallel, 3 cores</td>
<td>6474.010</td>
<td>6511.838</td>
<td>6543.365</td>
</tr>
<tr>
<td>parallel, 4 cores</td>
<td>5782.187</td>
<td>5812.958</td>
<td>5850.400</td>
</tr>
</tbody>
</table>

Table 18: Estimated parallelizable fraction of linear SVM hyperparameter search (AUTD experiment)

<table>
<thead>
<tr>
<th>$S(P)$</th>
<th>$P$</th>
<th>$\alpha$</th>
<th>Parallel fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.784</td>
<td>2</td>
<td>0.216</td>
<td>78.4%</td>
</tr>
<tr>
<td>2.435</td>
<td>3</td>
<td>0.283</td>
<td>71.7%</td>
</tr>
<tr>
<td>2.954</td>
<td>4</td>
<td>0.349</td>
<td>65.1%</td>
</tr>
</tbody>
</table>
Table 19: Estimated parallelizable fraction of linear SVM hyperparameter search (BCI competition)

<table>
<thead>
<tr>
<th>S(P)</th>
<th>P</th>
<th>α</th>
<th>parallel fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.914</td>
<td>2</td>
<td>0.086</td>
<td>91.4%</td>
</tr>
<tr>
<td>2.595</td>
<td>3</td>
<td>0.202</td>
<td>79.8%</td>
</tr>
<tr>
<td>2.907</td>
<td>4</td>
<td>0.364</td>
<td>63.6%</td>
</tr>
</tbody>
</table>

While the estimated parallelizable fraction of the SVM grid search task is lower than that of the SDA search, the overall fraction is still quite high.

### 6.5 Expected speedup in real processing

Processing speed is highly dependent on available hardware, but the above time and memory savings occur in key experimental tasks. Assume that a BCI processing chain consists of the following (memory/time independent) steps, whose total required time is unity:

1. Load data \((w)\)
2. Filter data \((x)\)
3. Preprocess/transform data \((y)\)
4. Tune and verify classifier \((z)\)

We can then describe the required time by the equation

\[
t_w + t_x + t_y + t_z = 1
\]

where \(t_w\) is the required proportionate time for task \(w\).

As given in the benchmark results, the non-parallelizable fraction of the filtering task \(x\) (for the first dataset) is 0.880, while the non-parallelizable fraction of the classification task \(z\) for a linear SVM is 0.283\(^{26}\). Additionally, the time to load the data comes from a single fixed step, and can also be neglected in the case of a local cluster or single machine. Thus, assuming that available computing grid capacity is large in comparison to the task overhead (a reasonable assumption in this case), the time required to process the experimental data used in the P300 case study is:

\[
t_{\text{parallel}} = 0.88t_x + 0.283t_z
\]

Assume further that the filtering step takes 10 percent of the total processing time, while hyperparameter tuning takes 90 percent. In this case, \(t_{\text{parallel}} = 34.3\%\). In other words, if processing took 60 minutes on a laboratory’s computing hardware with a non-parallelized program, this processing could be cut down to 24 minutes. The P300 experiment introduced in the case study contains data processed independently from 13 subjects. This means that if the data were available in advance, a BCI could be tuned for a single subject in \(\frac{24}{13} = 1.85\) minutes.

\(^{26}\)In practice, we found that the time required to compute the principal component analysis was negligible in comparison.
7 Summary

7.1 Main Results

The main contributions of this thesis lie in the unified, free/open source tool rBCI, which includes all of the optimizations introduced earlier. These optimizations were presented at the 2014 Asia-Pacific Signal and Information Processing Association [8]. Additionally, a variety of peer-reviewed libraries for data analysis and unsupervised learning were tested and incorporated into the final product, placing the contributions into a wider interdisciplinary academic context. The end result is a high performance, modular, standardized toolkit for brain-computer interface research and development accessible to specialists and non-specialists alike.

7.2 Summary of performance improvements

Performance improvements include: the introduction of parallelized processing and high-performance computing features in areas of BCI data processing identified to benefit most from distributed computing, the use of high-performance data structures to accelerate data access and limit memory overhead, and the unification of these and other features into a cross-platform, open-source R-based GUI enabling BCI researchers from all backgrounds to take advantage of the benefits of improved data processing, accelerating research output.

7.3 Summary of software toolkit features

On top of the improved data processing in the backend, the software toolkit rBCI consists of the following task modules:

- Data Import
- Exploration/Annotation
- Filtering
- Transformation/Unsupervised Learning
- Classification/Supervised Learning
- (Custom) Tool Design
- Performance Settings

Each of these modules functions independently of the others (excepting Data Import) and can be used to augment or replace existing research workflows.

7.4 Future Work

Development of rBCI does not end with this thesis. The software as developed is an evolving open-source project that we intend to support for an extended period of time. Its source code will be made freely available [7] for study, use, and outside contributions. Reasonable efforts will be made to incorporate significant processing/algorithm developments as they appear in the literature.
8 Appendix: Source code

Source code for all software and analytical findings given in this thesis is given alphabetically below, separated by file and additionally by section: Benchmarks for code used to benchmark performance, and rBCI for code implementing the functioning of the software named in the thesis.

The source code of rBCI is divided into frontend and backend scripts, and then subsequently divided by task. Smaller utility functions are also broken out into miscellaneous scripts where it improved the organization of the code. Backend scripts consist solely of function definitions to be imported into the environment namespace. Frontend scripts consist of GUI object definitions that create GUI objects on the fly and define behaviors such as calling backend functions. They consist of a single script that creates GUI objects for a task set, and scripts defining GUI objects for specific subtasks or child GUI objects for that task window. The main initialization script, init_interface.R, calls the backend initialization script init_backend.R, and instantiates the first GUI objects to present to the user.

For reference and attribution, libraries not mentioned explicitly in the thesis body but used in the code are cited here. [100][87][135][128][18][94][19][132][127][71][80][94][126][71]

8.1 Benchmarks

8.1.1 filtfiltbenchmark.R

```r
library(foreach)
library(iterators)
library(doMC)
library(signal)

setwd(project.root)
source('./.Rprofile')
# load unfiltered data
load('./data/AUTD/au/autd.Rdata')

setkey(erp,Trial,Sample)

# Filter out everything above 10 Hz
end.f <- 20
start.f <- 10
low.pass <- butter(buttord(Wp = start.f/sample.rate*2,
Ws = end.f/sample.rate*2,
Rp = 0.5, Rs = 40))

filter.para <- expression(erp[1:100000000,
foreach(this.trial=iter(erp[,unique(Trial)]),
.combine=c) %dopar% {
filtfilt(low.pass,erp[J(this.trial)
,Voltage])
})
```
filter.seri <- expression(erp[1:100000000, filtfilt(low.pass, Voltage),
  by=c("Trial", "Channel")])

### do benchmark

cycles <- 20
library(microbenchmark)
registerDoMC(cores=2)
filter.bench.2cores <- microbenchmark(list=c(filter.para, filter.seri),
  times=cycles)
registerDoMC(cores=3)
filter.bench.3cores <- microbenchmark(list=c(filter.para, filter.seri),
  times=cycles)
registerDoMC(cores=4)
filter.bench.4cores <- microbenchmark(list=c(filter.para, filter.seri),
  times=cycles)

## rename expression names for better formatting
new.levels <- as.factor(c("parallel, 2 cores", "serial"))
filter.bench.2cores$expr <- new.levels[filter.bench.2cores$expr]
new.levels <- as.factor(c("parallel, 3 cores", "serial"))
filter.bench.3cores$expr <- new.levels[filter.bench.3cores$expr]
new.levels <- as.factor(c("parallel, 4 cores", "serial"))
filter.bench.4cores$expr <- new.levels[filter.bench.4cores$expr]

## Merge different results into one table
filter.bench <-
  merge(x=filter.bench.2cores[
    which(filter.bench.2cores$expr ==
      levels(filter.bench.2cores$expr)[1])],
    y=filter.bench.3cores[
    which(filter.bench.3cores$expr ==
      levels(filter.bench.3cores$expr)[1])],
    all=TRUE)
filter.bench <-
  merge(x=filter.bench,
    y=filter.bench.4cores[
    which(filter.bench.4cores$expr ==
      levels(filter.bench.4cores$expr)[1])],
    all=TRUE)
filter.bench <-
  merge(x=filter.bench,
    y=filter.bench.4cores[
    which(filter.bench.4cores$expr ==
      levels(filter.bench.4cores$expr)[2])],
    all=TRUE)

### plot results
8.1.2 hyperoptbenchmark.R

```r
library(foreach)
library(iterators)
library(doMC)
library(signal)

setwd(project.root)
source('./.Rprofile')

##### DATA LOAD SECTION #####

this.subject <- "s13"

# # load filtered data
load("./data/AUTD/au/autd_filtered.Rdata")
setkey(erp.filtered,Subject)
this.erp <- erp.filtered[this.subject,]

this.erp <- baseline.remove(this.erp)

# Scale features
this.erp[,Voltage:= scale(Voltage)]

# train on two sessions, test on one
training.sessions <- c(1,2)
testing.sessions <- 3
setkey(this.erp,Session)
this.erp.training <- this.erp[J(as.factor(training.sessions))]
this.erp.testing <- this.erp[J(as.factor(testing.sessions))]

# this.erp.training <- trial.unit.averager(this.erp.training)
trial.sets <- simple.sampler(this.erp.training,equal.class=TRUE,train.size=1)

# Time subset
time.s <- 200
```
```r
time.e <- 450

# Channel subset
# chan.subset <- c(6,9)
chan.subset <- this.erp[, unique(Channel)]

this.erp.training <- trial.sets$training.table[(Time >= time.s &
  Time <= time.e &
  Channel %in% chan.subset),]
this.erp.testing <- this.erp.testing[(Time >= time.s &
  Time <= time.e &
  Channel %in% chan.subset),]

# Also produce channel-wise columnar tables for some learning algorithms
this.erp.training.channel <- channel.form(this.erp.training, has.dups = TRUE)
this.erp.testing.channel <- channel.form(this.erp.testing, has.dups = TRUE)

training.pca <- prcomp(~ . - Class-Trial-Time, data = this.erp.training.channel,
  scale = FALSE, tol = sqrt(.Machine$double.eps))

training.pca.data <- as.data.table(predict(training.pca))
training.pca.data[, Class := this.erp.training.channel[, Class]]
training.pca.data[, Trial := this.erp.training.channel[, Trial]]

testing.pca.pred <- predict(training.pca, newdata = this.erp.testing.channel)

testing.pca.pred <- as.data.table(testing.pca.pred)
testing.pca.pred[, Class := this.erp.testing.channel[, Class]]
testing.pca.pred[, Trial := this.erp.testing.channel[, Trial]]

testing.pca.long <- melt(testing.pca.pred,
  id.vars = c("Class", "Trial"))
setnames(testing.pca.long, old = colnames(testing.pca.long),
  new = c("Class", "Trial", "PC", "value"))

shrinkage.subset <- list(lambda = seq(from = 0, to = 0.5, by = 0.01),
  lambda.var = seq(from = 0, to = 0.01, by = 0.001))
sda.opt.exp <- expression(
  foreach(this.lambda = iter(shrinkage.subset$lambda),
    .combine = cbind) %:%
  foreach(this.lambda.var = iter(shrinkage.subset$lambda.var),
    .combine = rbind))

shrinkage.subset <- list(lambda = seq(from = 0, to = 0.5, by = 0.01),
  lambda.var = seq(from = 0, to = 0.01, by = 0.001))
sda.opt.exp <- expression(
  foreach(this.lambda = iter(shrinkage.subset$lambda),
    .combine = cbind) %:%
  foreach(this.lambda.var = iter(shrinkage.subset$lambda.var),
    .combine = rbind))

# print(this.time.interval)
# print(this.chan)
```
sda.model <- sda(lambda.freqs = 1, # uniform prior
    verbose=FALSE, # for benchmarking
    lambda=this.lambda,
    lambda.var=this.lambda.var,
    Xtrain=as.matrix(training.pca.data[,1:3,with=FALSE]),
    L=training.pca.data[,as.factor(Class)])

sda.pred <- predict(verbosel=FALSE, # for benchmarking
    sda.model,as.matrix(testing.pca.pred[,1:3,with=FALSE]))

    # auc(roc(predictions = sda.pred$class, # labels = testing.pca.pred[,as.factor(Class)]))

acc <- sum(sda.pred$class != testing.pca.pred[,Class]) /
    testing.pca.pred[,length(Class)]

if (acc < 0.5) { acc <- 1 - acc}

acc

# do benchmark
cycles <- 20
library(microbenchmark)

 exp.core1 <- expression({
    registerDoMC(cores=1)
    foreach(this.lambda=iter(shrinkage.subset$lambda),
        .combine=cbind) %:%
    foreach(this.lambda.var=iter(shrinkage.subset$lambda.var),
        .combine=rbind) %dopar% {
        # print(this.time.interval)
        # print(this.chan)

    sda.model <- sda(lambda.freqs = 1, # uniform prior
        verbose=FALSE, # for benchmarking
        lambda=this.lambda,
        lambda.var=this.lambda.var,
        Xtrain=as.matrix(training.pca.data[,1:3,with=FALSE]),
        L=training.pca.data[,as.factor(Class)])

    sda.pred <- predict(verbosel=FALSE, # for benchmarking
        sda.model,
        as.matrix(testing.pca.pred[,1:3,with=FALSE]))

    # auc(roc(predictions = sda.pred$class, # labels = testing.pca.pred[,as.factor(Class)]))

})

# do benchmark

# labels = testing.pca.pred[,as.factor(Class)])

acc <- sum(sda.pred$class != testing.pca.pred[,Class]) / testing.pca.pred[,length(Class)]

if (acc < 0.5) { acc <- 1 - acc }

acc

}

exp.core2 <- expression({
  registerDoMC(cores=2)
  foreach(this.lambda=iter(shrinkage.subset$lambda),
    .combine=cbind) %:%
  foreach(this.lambda.var=iter(shrinkage.subset$lambda.var),
    .combine=rbind) %dopar%
    {
      # print(this.time.interval)
      # print(this.chan)

      sda.model <- sda(lambda.freqs = 1, # uniform prior
                      verbose=FALSE, # for benchmarking
                      lambda=this.lambda, 
                      lambda.var=this.lambda.var, 
                      Xtrain=as.matrix(training.pca.data[,1:3,with FALSE]), 
                      L=training.pca.data[,as.factor(Class)])

      sda.pred <- predict(verbos=FALSE, # for benchmarking
                           sda.model, 
                           as.matrix(testing.pca.pred[,1:3,with FALSE]))

      # auc(roc(predictions = sda.pred$class, 
      # labels = testing.pca.pred[,as.factor(Class)]))

      acc <- sum(sda.pred$class != testing.pca.pred[,Class]) / testing.pca.pred[,length(Class)]

      if (acc < 0.5) { acc <- 1 - acc }

      acc
    }
  })

exp.core3 <- expression({
  registerDoMC(cores=3)
  foreach(this.lambda=iter(shrinkage.subset$lambda),
    .combine=cbind) %:%
  foreach(this.lambda.var=iter(shrinkage.subset$lambda.var),
    .combine=rbind) %dopar%
    {
      # print(this.time.interval)
# print(this.chan)

sda.model <- sda(lambda.freqs = 1, # uniform prior
               verbose=FALSE, # for benchmarking
               lambda=this.lambda,
               lambda.var=this.lambda.var,
               Xtrain=as.matrix(training.pca.data[,1:3,with=FALSE]),
               L=training.pca.data[,as.factor(Class)])

sda.pred <- predict(verb<>s FALSE, # for benchmarking
               sda.model,
               as.matrix(testing.pca.pred[,1:3,with=FALSE]))

# auc(roc(predictions = sda.pred$class,
# labels = testing.pca.pred[,as.factor(Class)]))

acc <- sum(sda.pred$class != testing.pca.pred[,Class]) /
       testing.pca.pred[,length(Class)]

if (acc < 0.5) { acc <- 1 - acc}

acc

})

exp.core4 <- expression({
  registerDoMC(cores=4)
  foreach(this.lambda=iter(shrinkage.subset$lambda),
          .combine=cbind) %:%
    foreach(this.lambda.var=iter(shrinkage.subset$lambda.var),
            .combine=rbind) %dopar% {
      # print(this.time.interval)
      # print(this.chan)

    sda.model <- sda(lambda.freqs = 1, # uniform prior
                     verbose=FALSE, # for benchmarking
                     lambda=this.lambda,
                     lambda.var=this.lambda.var,
                     Xtrain=as.matrix(training.pca.data[,1:3,with=FALSE]),
                     L=training.pca.data[,as.factor(Class)])

    sda.pred <- predict(verb<>s FALSE, # for benchmarking
                        sda.model,
                        as.matrix(testing.pca.pred[,1:3,with=FALSE]))

    # auc(roc(predictions = sda.pred$class,
    #           labels = testing.pca.pred[,as.factor(Class)]))

    acc <- sum(sda.pred$class != testing.pca.pred[,Class]) /

testing.pca.pred[, length(Class)]

if (acc < 0.5) { acc <- 1 - acc }
acc
}

sda.opt.bench <- microbenchmark(list=c(exp.core1, exp.core2, exp.core3, exp.core4),
times=cycles)
new.levels <- as.factor(c("serial", "parallel, 2 cores", "parallel, 3 cores", "parallel, 4 cores"))
sda.opt.bench$expr <- new.levels[sda.opt.bench$expr]
names(sda.opt.bench) <- c("Type", "Time")
sda.opt.bench$Type <- relevel(sda.opt.bench$Type, "serial")

library(ggplot2)
qplot(y=Time, x=Type, data=sda.opt.bench, geom="boxplot",
     xlab="Type", ylab="Time(s)",
     fill=Type)

library(LiblineaR)
class.weights <- c("0"=0.4, "1"=0.4)
# costs.subset <- seq(from = 420, to = 580, by = 10)
costs.subset <- seq(from= 400, to = 600, by= 20)

linsvm.opt.exp <- expression(

  foreach(this.cost=iter(costs.subset), .combine=rbind) {%
    foreach(this.run=iter(seq_len(10)), .combine=cbind) {%
      svm.model <- LiblineaR(wi = class.weights, cost = this.cost, type=3, # cross=3,
data = training.pca.data[,1:3,with=FALSE],
labels = training.pca.data[,Class])

svm.pred <- predict(svm.model,
        newx=testing.pca.pred[,1:3,with=FALSE])

acc <- sum(svm.pred$predictions != testing.pca.pred[,Class])/
        length(svm.pred$predictions)

if (acc < 0.5) { acc <- 1 - acc}

acc

exp.core1 <- expression({
    registerDoMC(cores=1)
    foreach(this.cost=iter(costs.subset),
        .combine=rbind) %:%
        foreach(this.run=iter(seq_len(10)), .combine=cbind) %dopar% {
#        print(this.time.interval)
#        print(this.chan)

    svm.model <- LiblineaR(wi = class.weights,
                cost = this.cost,
                type=3, # cross=3,
                data = training.pca.data[,1:3,with=FALSE],
                labels = training.pca.data[,Class])

    svm.pred <- predict(svm.model,
        newx=testing.pca.pred[,1:3,with=FALSE])

    acc <- sum(svm.pred$predictions != testing.pca.pred[,Class])/
            length(svm.pred$predictions)

    if (acc < 0.5) { acc <- 1 - acc}

    acc
    }
})

exp.core2 <- expression({
    registerDoMC(cores=2)
    foreach(this.cost=iter(costs.subset),
        .combine=rbind) %:%
        foreach(this.run=iter(seq_len(10)), .combine=cbind) %dopar% {
#        print(this.time.interval)
#        print(this.chan)

    svm.model <- LiblineaR(wi = class.weights,
                cost = this.cost,
                type=3, # cross=3,
                data = training.pca.data[,1:3,with=FALSE],
                labels = training.pca.data[,Class])

    svm.pred <- predict(svm.model,
        newx=testing.pca.pred[,1:3,with=FALSE])

    acc <- sum(svm.pred$predictions != testing.pca.pred[,Class])/
            length(svm.pred$predictions)

    if (acc < 0.5) { acc <- 1 - acc}

    acc
    }
})
```
svm.model <- LiblineaR(
  wi = class.weights,
  cost = this.cost,
  type=3, # cross=3,
  data = training.pca.data[,1:3,with=FALSE],
  labels = training.pca.data[,Class])

svm.pred <- predict(svm.model,
  newx=testing.pca.pred[,1:3,with=FALSE])

acc <- sum(svm.pred$predictions != testing.pca.pred[,Class])/
  length(svm.pred$predictions)

if (acc < 0.5) { acc <- 1 - acc}
acc
}
}

exp.core3 <- expression(
  registerDoMC(cores=3)
  foreach(this.cost=iter(costs.subset),
    .combine=rbind) %:%
  foreach(this.run=iter(seq_len(10)), .combine=cbind) %dopar% {
    svm.model <- LiblineaR(
      wi = class.weights,
      cost = this.cost,
      type=3, # cross=3,
      data = training.pca.data[,1:3,with=FALSE],
      labels = training.pca.data[,Class])

    svm.pred <- predict(svm.model,
      newx=testing.pca.pred[,1:3,with=FALSE])

    acc <- sum(svm.pred$predictions != testing.pca.pred[,Class])/
      length(svm.pred$predictions)

    if (acc < 0.5) { acc <- 1 - acc}
    acc
  }
)

exp.core4 <- expression(
  registerDoMC(cores=4)
  foreach(this.cost=iter(costs.subset),
    .combine=rbind) %:%
```

foreach(this.run=iter(seq_len(10)), .combine=cbind) %dopar% {
    #
    print(this.time.interval)
    #
    print(this.chan)

    svm.model <- LiblineaR( wi = class.weights, 
        cost = this.cost, 
        type=3, # cross=3, 
        data = training.pca.data[,1:3,with=FALSE], 
        labels = training.pca.data[,Class])

    svm.pred <- predict(svm.model, 
        newx=testing.pca.pred[,1:3,with=FALSE])

    acc <- sum(svm.pred$predictions != testing.pca.pred[,Class])/
            length(svm.pred$predictions)

    if (acc < 0.5) { acc <- 1 - acc}

    acc

}

linsvm.opt.bench <- microbenchmark(list=c(exp.core1, 
        exp.core2, 
        exp.core3, 
        exp.core4), 
        times=cycles)

new.levels <- as.factor(c("serial","parallel, 2 cores", 
                        "parallel, 3 cores", 
                        "parallel, 4 cores"))
linsvm.opt.bench$expr <- new.levels[linsvm.opt.bench$expr]
names(linsvm.opt.bench) <- c("Type","Time")
linsvm.opt.bench$Type <- relevel(linsvm.opt.bench$Type,"serial")

library(ggplot2)
qplot(y=Time, x=Type, data=linsvm.opt.bench, geom="boxplot", 
    xlab="Type", ylab="Time(s)", 
    fill=Type)

8.1.3 memorybenchmarks.R

library(lineprof) # memory profiling
library(signal) # signal processing
library(ggplot2) # plotting
library(knitr) # formatting
library(plyr) # non call-by-reference split-apply-combine functions

setwd(project.root)
source('./.Rprofile')
# load unfiltered data
load('./data/AUTD/au/autd.Rdata')

freq.bins <- 500

# limit the number of points we evaluate for speed
test.table <- erp[Trial < 100,]
# print size of test object
print(object.size(test.table),units="MB")

##### memory cost of grouped average #####
mean.mem.prof.ref <- lineprof(test.table[,mean(Voltage),
                                by=c("Sample","Channel","Class"))
mean.mem.prof.def <- lineprof(ddply(test.table,.(Sample,Channel,Class),
                                  summarize, meanv=mean(Voltage)))

# View profiler output - call by reference
shine(mean.mem.prof.ref)
# View profiler output - call by value
shine(mean.mem.prof.def)

# Clean up
rm(mean.mem.prof.def)
rm(mean.mem.prof.ref)

erp.trial.mean <- erp[,mean(Voltage),
                       by=c("Sample","Channel","Class")]

##### memory cost of computing channel spectra #####
spectra.mem.prof.ref <- lineprof(erp.trial.mean[,get.spectrum(V1),
                                           by=c("Channel","Class"))
spectra.mem.prof.def <- lineprof(ddply(erp.trial.mean,.
                                        (Channel,Class), summarize,
                                        spec=get.spectrum(V1)))

# View profiler output - call by reference
shine(spectra.mem.prof.ref)
# View profiler output - call by value
shine(spectra.mem.prof.def)

##### windowed average benchmark #####
window.avg.mem.prof.ref <- lineprof(
test.table[, roll_mean(Voltage, n = 10),
  by = c("Subject", "Trial", "Channel"))
window.avg.mem.prof.def <- lineprof(
  ddply(test.table, .(Subject, Trial, Channel),
    summarize,
    wavg = roll_mean(Voltage, n=10)))
shine(window.avg.mem.prof.ref)
shine(window.avg.mem.prof.def)

# Filter out everything above 10 Hz
library(signal)
end.f <- 20
start.f <- 10
low.pass <- butter(buttord(Wp = start.f/sample.rate*2,
  Ws = end.f/sample.rate*2,
  Rp = 0.5, Rs = 40))
erp.filtered <- copy(erp)
test.table.filt <- copy(test.table)

##### memory cost of filtering signal #####
erp.filtered <- erp.filtered[, Voltage:= filtfilt(low.pass, Voltage),
  by = c("Trial", "Channel")]
filter.mem.prof.ref <- lineprof(test.table.filt[, Voltage:= filtfilt(low.pass, Voltage),
  by = c("Trial", "Channel")])
filter.mem.prof.def <- lineprof(ddply(test.table.filt, .(Trial, Channel),
  mutate,
  Voltage:= filtfilt(filt=low.pass, x=Voltage)))
shine(filter.mem.prof.def)
shine(filter.mem.prof.ref)

# filt.test <- ddply(test.table.filt, .(Trial, Channel),
#   mutate,
#   Voltage:= filtfilt(filt=low.pass, x=Voltage))
##### benchmark scaling signal #####

```r
this.subject <- "s13"

# load unfiltered data
# load("./data/AUTD/au/autd.Rdata")

this.erp <- erp[Subject == this.subject,]

# # load filtered data
load("./data/AUTD/au/autd_filtered.Rdata")
setkey(erp.filtered,Subject)
this.erp <- erp.filtered[this.subject,]

# Scale features
scale.mem.prof.ref <- lineprof(this.erp[,Voltage := scale(Voltage)])
scale.mem.prof.def <- lineprof(this.erp$Voltage <- scale(this.erp$Voltage))

# neither call results in copying
```

8.2 rBCI

8.2.1 apply_filter.R

```r
apply.filter <- function(signal.table, filt.groups, val.col, filter.obj) {

    ## Parallelization: assume first group is largest, use foreach() and keyed
    ## table

    table.filtered <- copy(signal.table)
    setkeyv(table.filtered, filt.groups[[1]])

    if (getDoParRegistered()) { # parallel case
        foreach(
            this.group =
            table.filtered[J(this.group),
                unique(get(filt.groups[[1]])]) %dopar%
                c(val.col) :=
                filtfilt(filter.obj,
                    get(val.col)),
                by = c(filt.groups[[2]]))
        } else {
            table.filtered <-
        }
```
table.filtered[, c(val.col) := filtfilt(filter.obj, get(val.col)),
    by = filt.groups]
}
return(table.filtered)
### TODO examine reference dispatch in distributed context

8.2.2 classifiers.R

### SVM functions

train.svm.model <- function(train.data,
    kern.type, target.col, feature.cols, cost.param, kern.params = list(), ...)
{
    ## sanitize params
    cost.param <- if(is.na(as.numeric(cost.param)) || # R standard null coalescing op
        !(as.numeric(cost.param) > 0)) 1 else cost.param

    if (kern.type == "Linear") {
        ## use LiblineaR
        ## assume uniform prior for now
        class.labels <- train.data[, levels(factor(get(target.col)))]
        weights <- rep(1, times = length(class.labels))/length(class.labels)
        weights <- setNames(weights, c(class.labels))
        svm.model <-
            LiblineaR(wi = weights,
                cost = cost.param,
                type = 3, # TODO comment this
                data = train.data[, feature.cols, with=FALSE],
                labels = train.data[,factor(get(target.col))])
    } else {
        ## use kernlab
        ## map kernel types to param strings
        switch(kern.type,
            "Laplace"=" { kern.type <- "laplacedot" },
            "Gaussian"=" { kern.type <- "rbfdot" },
            "Polynomial"=" { kern.type <- "polydot" },
            "Hyperbolic"=" { kern.type <- "tanhdot" },
            "Bessel"=" { kern.type <- "besseldot" },
        )
    }
}
"ANOVA RBF"="{ kern.type <- "anovadot" },
"Spline"="{ kern.type <- "splinedot" }
"

## make params list
## param.list <-

svm.model <-
ksvm(data = train.data,
    kernel = kern.type,
    C = cost.param,
    kpar = kern.params,
    x = as.formula(paste(target.col,"~", feature.cols)))

}  
return(svm.model)
}

test.svm.model <- function(test.data, svm.model, feature.cols) {
## all decent SVM packages overload predict(), but just in case this is for
## portability

predict(svm.model,
    test.data[,feature.cols, with = FALSE])
}

plot.svm.model <- function(svm.model, ...) {
## LiblineaR doesn't have a plot method, so check the model type
if (class(svm.model) == "LiblineaR") {
### TODO make this more consistent
    return(NULL)
}

return(plot(svm.model))
}

table.svm.model <- function(svm.prediction, test.data) {
  switch(class(svm.prediction),
    "list"="{ # predict.LiblineaR() returns a list
            table(prediction = svm.prediction$predictions,
                  actual = test.data$Class)
      },
    "matrix"="{ # predict.ksvm() returns a matrix
                  table(prediction = svm.prediction,
                        actual = test.data)
      }
  )
}
### SDA functions

```r
train.sda.model <- function(train.data,
    target.col,
    feature.cols,
    sda.lambda,
    sda.lambda.var) {

    sda.model <-
        sda(lambda.freqs = 1, # uniform prior
            as.matrix(train.data[, c(feature.cols), with = FALSE]),
            train.data[,get(target.col)],
            lambda = sda.lambda,
            lambda.var = sda.lambda.var)
}

test.sda.model <- function(sda.model, test.data, feature.cols) {
    ### TODO error checking on input

    predict(sda.model, as.matrix(test.data[, feature.cols, with=FALSE]))
}

# TODO sda does not have a native plot function

table.sda.model <- function(sda.prediction, test.data) {

    table(predicted = sda.prediction$class,
          data = test.data$Class)
}

### Bayes functions

train.bayes.model <- function(train.data,
    bayes.smooth,
    target.col,
    feature.cols) {

    # validate target as factor type
    train.data[,c(target.col) := factor(get(target.col))]

    bayes.model <-
        naiveBayes(formula = as.formula(paste(target.col,"~", feature.cols)),
                    data = train.data,
                    laplace = bayes.smooth)
}
```
test.bayes.model <- function(bayes.model, test.data) {
  bayes.pred <- list(prediction = predict(bayes.model, test.data))
  return(bayes.pred)
}

### TODO bayes does not have a native plot function

table.bayes.model <- function(bayes.prediction, test.data) {
  table(bayes.prediction$prediction, test.data)
}

8.2.3 dependencies.R

### libraries required
## should be consolidated here like a package.json file in node

cran.dependencies = c(
  "gWidgets", # GUI framework
  "gWidgetsRGtk2", # bindings for GTK2
  "tools",
  "devtools", # for installing github libs
  "parallel", # TODO needed?
  "foreach", # explicit parallel frontend
  "doMC", # local multicore backend
  "doSNOW", # SNOW cluster backend
  "doMPI", # MPI cluster backend
  "doRedis", # Redis job store backend
  "R.matlab", # for importing matfiles
  "signal", # signal processing lib
  "abind", # multivariate array concatenator
  "R.utils",
  "reshape2", # for dcast, melt
  "gdata", # for cleanups: keep"
  "data.table", # hyperfast memory-efficient data struct
  "ggplot2", # general plotting engine
  "JADE", # joint diagonalization
  "psych", # geometric, harmonic means
  "cluster", # clusplot, silhouette
  "sda", # sda classifier
  "LiblineaR", # linear svm classifier
  "kernlab", # kernel pca/svm lib
)
"e1071" , # naiveBayes classifier
"knitr"     # report generator

github.dependencies = list(
  ggbplot = "vqv/ggbplot" # ggidevtools-dependent
)

## install and load missing dependencies
lapply(cran.dependencies, function(x) {
  if (!require(x, character.only = TRUE)) {
    install.packages(x)
  }

  library(x, character.only = TRUE)
})

lapply(github.dependencies, function(x) {
  package.basename <- eval.parent(quote(names(X)))[substitute(x)[[3]]]

  if (!require(package.basename, character.only = TRUE)) {
    install_github(x)
  }

  ## print(package.basename)
  library(package.basename, character.only = TRUE)
})

8.2.4 downsample_dt.R

```r
##### time-series downsampling #####
downsample.dt <- function(input.table, time.col = "Time", ds.factor = 0.5) {
  # sort first
  setkeyv(input.table,time.col)
  
  # produce indices of proper downsampled interval length
  resample.seq <- seq(to=input.table[,length(unique(get(time.col))))], by = floor(1/ds.factor),
  length.out = ceiling(ds.factor*input.table[,length(unique(get(time.col))))])]

  # convert to existing values for lookup
  resample.vals <- input.table[,unique(get(time.col))][resample.seq]

  return(input.table[get(time.col) %in% resample.vals,])
}
```
### Initialize data annotation tags for backend processing

```r
# TODO change this from bulk init to on-demand by-file init
# rbci.env$tags[names(rbci.env$importlist)] <- list(rbci.env$taglist)

summarize <- function(eeg.table, selected.columns) {
  return(capture.output(
    summary(rbci.env$importlist[[1]][,c(2,3),with=FALSE])
    summary(eeg.table[,selected.columns,with=FALSE])
  ))
}
```

### plotting function for multivariate grand means

# first arg is time column
# second arg is channel column
# third arg is class column
# http://stackoverflow.com/a/10659563/2023432

define_grand_means_plot <- function(eeg.table, val.name = "Voltage", time.name = "Sample", chan.name = "Channel", targ.name = "Class", plot.title = NULL) {
  title.caption <- "Averaged ERP by Class"
  if (!is.null(plot.title)) {
    title.caption <- paste(title.caption, "\n", plot.title, sep = "\n")
  }
  comb.class.avg <- eeg.table[,mean(get(val.name)), by = c(time.name, chan.name, targ.name)]
  comb.class.avg[, targ.name := as.factor(get(targ.name)),
    with = FALSE]
  setnames(comb.class.avg, old=colnames(comb.class.avg),
    new=c(colnames(comb.class.avg)[1:length(colnames(comb.class.avg))-1], val.name))

  # checkplot
  preview.plot <- ggplot(comb.class.avg,
    aes_string(time.name, val.name, 
      label = targ.name, group = targ.name)) +
geom_line(aes_string(colour = targ.name )) +
stat_smooth(aes_string(colour = targ.name),
  method = "loess", level=0.9) +
facet_wrap(as.formula(paste("~", chan.name)), ncol=4) +
ggtitle(title.caption) +
xlab(time.name) + ylab("Amplitude (uV)") +
## guides(col = guide_legend(nrow = 28, byrow=TRUE,
##       title = "Channel")) +
theme(plot.title = element_text(size = 18, face = "bold",
  colour = "black", vjust=1))

preview.plot

hist.plot <- function(data.set) {
  data.melt <- melt(data.set)

data.plot <-
  ggplot(data.melt, aes(x = value, color = variable)) +
  facet_wrap(~variable, scales = "free") +
  geom_density()

  return(data.plot)
}

8.2.6 importers.R

```r
#' Import functions for external data files.
#'
#' Each of these functions imports data into a specified environment.
#'
#' @param filename A string pointing to the data file.
#' @param environment The desired environment to receive the data.
#' @param options A list of named options
#' @return If import OK, outputs a preview of the imported data
#' @name Import backends
#' @rdname matlab.type2.import

matlab_type2_import <- function(init.struct, eeg.ind, tgt.ind, preview = FALSE) {
  # TODO remove structural hardcoding to frontend
  init.eeg <- init.struct$eeg[[eeg.ind]]
  init.tgt <- init.struct$eeg[[tgt.ind]]

  eeg.table <- melt(init.eeg,
                   varnames = c("Trial", "Sample", "Channel"),
                   value.name = "Voltage")
```
### Initialize backend processing #######

```r
cell(1)
## load libs
source("./backend/dependencies.R")

cell(2)
## load data structures
source("./backend/structures.R")

cell(3)
## misc utility functions
source("./backend/misc.R")

### importer backend
source("./backend/importers.R")

### explorer backend
source("./backend/explorers.R")

### filter backend
source("./backend/simple_filter.R")
source("./backend/plot_filter.R")
source("./backend/apply_filter.R")
source("./backend/downsample_dt.R")

### transformer backend
source("./backend/transformers.R")

### classifier backend
source("./backend/classifiers.R")

### reporter backend
source("./backend/reporter.R")
```

### define special concatenating function along fourth dimension for files
```r
cell(1)
myabind <- function(x,y) {
  abind(x,y,along=4)
}
```

### curried abind for foreach; binds its arguments along 3rd dimension
abind3curry <- function(...) {
    abind(..., along=3)
}

### readRDS()/unserialize() doesn't work right now, so use RData and envs
### http://stackoverflow.com/a/5577647/2023432
load_obj <- function(f)
{
    temp.env <- new.env()
    nm <- load(f, envir = temp.env)[1]
    temp.env[[nm]]
}

### pseudo-curry of gwidgets enabled() method to enable bulk enable/disable
### widgets
enabled.list <- function(state = TRUE, ...) {
    require(gWidgets)
    ### TODO throw error if state not T/F
    ### TODO throw error if not passed good objs
    objs <- list(...)

    lapply(objs, function(x) {
        enabled(x) <- state
    })
}

### hacky widget refresher, destroys and recreates widgets
refresh.widget <- function(container, old.widget, new.widget = old.widget) {
    delete(container, old.widget)
    add(container, new.widget)
}

scoot.gtable.row <- function(input.gtable, input.row, direction = "up") {
    ### Scoots rows up or down in a data.frame-like gtable or gdf
    ### Relies on cool frame-like coercing of gtable[]
    ### TODO unique row checking/exception
    ### TODO error checking on table ends (fails silently currently)
    require(gWidgets)

    ## what kind of row arg did we get? a row or a row index?
    switch(class(input.row),
        "data.frame" = {
            # if we got a row, scan to find its index
            my.ind <- rowscan.data.frame(input.gtable, input.row)
        },
        "integer" = {
            # if we got an index, pull the row
    )
my.ind <- input.row
input.row <- input.gtable[input.row,]
}

## pick out the swapping buddy
switch(direction,
   "up" = {
      row.ind <- if (my.ind-1 > 0) my.ind-1 else my.ind
      swap.row <- input.gtable[row.ind,]
   },
   "down" = {
      row.ind <- if (my.ind+1 > length(input.gtable)) my.ind+1 else my.ind
      swap.row <- input.gtable[row.ind,]
   } )

## make the swap
input.gtable[row.ind,] <- input.row
input.gtable[my.ind,] <- swap.row

## no returns should be necessary, since changes are by ref
### TODO use later for error handling
return()
}

rowscan.data.frame <- function(df, row) {
### finds the row index(es) of a given row in a given dataframe
### from https://stat.ethz.ch/pipermail/r-help/2010-November/261170.html

## first candidate: fails on type issues
## rows <- which(apply(mapply(df, row, FUN="=="), MARGIN=1, FUN=all))

## string search: works, might be slow
### TODO revisit this
which((apply(df, 1, toString) %in% toString(row)))
}

### quick correlation matrix function for data.tables using special dcast()
acorr.table <- function(input.table, time.col, chan.col, val.col, trial.col) {
require(data.table)
require(foreach)

## in case we haven't been given something properly keyed
setkeyv(input.table, trial.col, chan.col, sample.col)
corr.dt <- foreach(seq_len(max(get(trial.col)))),
   .inorder = FALSE,
   .combine = abind3curry,
   .multicombine = TRUE) %dopar% {


cor(dcast.data.table(input.table[J(get(trial.col))]),
    as.formula(paste(time.col,"~",chan.col)),
    value.var = val.col)
}

8.2.9 partition_table.R

partition.table <- function(table.data, part.col, proportions, part.type) {
  ### TODO error checking

  setkeyv(table.data, part.col) # speedup

  switch(part.type,
         "Sequential"="{
            seq <- table.data[,unique(get(part.col))]
        }
         "Random"="{
            seq <- sample(table.data[,unique(get(part.col))])
        }
  )

  seq.parts <- subset.seq(seq, proportions)

  ## seq.parts is a list which we can use with lapply()
  ## TODO review this function for memory use
  lapply(seq.parts, function(x) {
    table.data[J(x)]
  })

  ## the easy part
  mapply(function(x,y) {
    sequence[x:y]
  }, slice.starts, slice.ends)

### returns a list

```r
plot.filter <- function(filter, sample.rate = 1, active.band) {
  require(ggplot2)

  filter.char <- freqz(filter, Fs = sample.rate)
  filter.char <- data.frame(freq = filter.char$f,
                            # conversion to decibels
                            mag = 20 * log10(abs(filter.char$h)))
  ## remove numerically erroneous values
  filter.char[which(is.infinite(filter.char$mag)), ] <- NA
  ## make plotting window make sense
  active.band <- sort(active.band)
  plot.freq.band <- c(max(c(active.band[[1]] - 10, 0)), # 10Hz margin between
                      min(c(active.band[[2]] + 10, sample.rate)))
  plot.mag.band <- c( # match the vertical axis by index of horizontal
                     filter.char$mag[[max(which(filter.char$freq <= active.band[[1]]))]],
                     filter.char$mag[[min(which(filter.char$freq >= active.band[[2]]))]])

  ggplot(filter.char, aes(x = freq, y = mag)) +
  scale_x_continuous(limits = range(plot.freq.band)) +
  scale_y_continuous(limits = range(plot.mag.band)) +
  geom_line() +
  ggtitle(bquote("Magnitude Response of Applied Filter")) +
  xlab("Frequency (Hz)") +
  ylab("Amplitude (dB)")
```

---

8.2.11 previewers.R

```r
#' Preview functions for external data files.
#' #'
#' # Each of these functions generates a text preview of a possible import, using
#' #'
#' @param filename A string pointing to the data file.
#' #'
#' @param environment The desired enviroment to receive the data.
```
add.step <- function(func.name, step.args) {
### adds called analysis steps and their args to the reporter view.
### Should be called by every analysis GUI button.

## a short summary for the user
## TODO first arg of all main analysis functions should be the dataset
## abbreviate the first argument call to 50 chars in case it’s huge
main.arg.short <- substr(as.character(step.args[1]), 1, 50)
step.summary <- paste(func.name, "of", main.arg.short)

### we assume arguments are explicitly named for all functions of this kind
### see https://github.com/talexand/rbci/issues/55 for discussion
new.step <- list(summary = step.summary,
                 enabled = FALSE,
                 code = paste(deparse(
                   bquote(do.call(.)(func.name), .(step.args)),
                   # ensure we get knittable code
                   control = c("showAttributes" = NULL),
                   collapse = "\\n"))
)

rbci.env$steplist <- append(rbci.env$steplist,
                           list(new.step))

### converts steplist to table form for GUI interaction
### (once done, the report generator also has to use this form, but it’s only
### done once)
if (length(steplist) > 0) {
  t(sapply(steplist, unlist))
} else {
  ## return a dummy df that matches structure
  ## TODO revisit this
  return(t(as.matrix(list(summary.text="", code="", enabled=""))))
}

### toggles enabled.col of steplist.table
steplist.table[row.ind,]$enabled <-
  !as.logical(steplist.table[row.ind,"enabled"])

### sanitize output.dir (gWidgets gives single quotes, against convention?)
output.dir <- gsub("\\", "", output.dir)

## collect up a frame of code objects that need to be run to be passed out
step.list <- as.data.frame(
  steplist.table[which(steplist.table[,"enabled"] == TRUE),]
)

## validation: fail if no steps enabled
if (nrow(step.list) == 0) { return("no steps to run") }

```r
## save environment to data directory
env.file.path <-
paste(output.dir, "/", strsplit(report.title, " ")[[1]][1],
  ".RData", # needed?
  sep = "")
save(rbci.env,
  file = env.file.path)
## make relative name for portability
env.file.name <- paste("./", strsplit(report.title, " ")[[1]][1],
  ".RData", sep = "")
## copy backend files to data directory
### TODO make this more portable/track package changes
dir.create(paste(output.dir, "/backend", sep = ""), mode = "0775")
file.copy(from = "/backend",
          to = paste(output.dir, "/", sep = ""),
          recursive = TRUE)
## get an R Markdown file ready
file.name <- paste(output.dir, "/", strsplit(report.title, " ")[[1]][1],
  ".Rmd", # needed?
  sep = "")
file.create(file.name)

file.conn <- file(file.name, open = "a")
## write header junk like title and author
writeLines(make.report.head(report.title, report.author), file.conn)
## write environment+backend load line
writeLines(c("# {r, results="hide", message=FALSE}'
       , 
       'source("./backend/dependencies.R")'
       , 
       'sourceDirectory("./backend")'
       , 
       paste("load("env.file.name", envir = .GlobalEnv")"),
       n Echo = "")
       ,
       file.conn)
apply(step.list, 1, function(this.step) { # go down by rows (of the df)
  ## write summary
  writeLines(c("#", this.step["summary"],
             "# {r}" , # write RMarkdown delimiters and code
             this.step["code"],
             n Echo = "")
             ,
             con = file.conn)
})
close(file.conn)
```
## call knitr on resulting Rmd file if enabled

```r
if (knit.now == TRUE) {

    ## do some directory finagling to make sure knitr is happy
    cur.dir <- getwd()
    setwd(output.dir)

    knit2html(basename(file.name))

    setwd(cur.dir) # undo dir finagle

    if (interactive()) {
        browseURL(paste("file://", file_path_sans_ext(file.name), ".html", sep = ""))
    }
}
```

```r
make.report.head <- function(report.title, report.author) {
    ### convenience function for generating a decent rmarkdown header
    ### TODO add settings
    ### TODO add backend function importer (for now import all funcs; show=false)

    c(paste(report.title, "\n","======","\n", sep=""),
    paste("## ", report.author, "\n", sep=""),
    paste("*", Sys.time(), "*", "\n", sep=""))
}
```

### simple_filter.R

```r
simple.filter <- function(filter.type,
                        active.band,
                        sample.rate,
                        filter.groups) {

    # filter.order <- 8 # TODO replace this with selectable order later
    band.tolerance <- 10 # Hz margin between pass/stop

    switch(filter.type,
            Lowpass = {
                pass.band <- active.band[[2]] # for the one-sided filters we use
       # only one frequency

                stop.band <- pass.band + band.tolerance
            },
            Highpass = {
                pass.band <- active.band[[1]]
            }
```
\[ \text{Stopband} = \{ \]
\[ \text{stop.band} \leftarrow \text{active.band} \]
\[ \text{pass.band} \leftarrow c(\text{stop.band}[1] - \text{band.tolerance}, \]
\[ \text{stop.band}[2] + \text{band.tolerance}) \]
\[ \}, \]
\[ \text{Bandpass} = \{ \]
\[ \text{pass.band} \leftarrow \text{active.band} \]
\[ \text{stop.band} \leftarrow c(\text{pass.band}[1] - \text{band.tolerance}, \]
\[ \text{pass.band}[2] + \text{band.tolerance}) \]
\[ \} \]
\[ \]
\[ \text{# designed.filter} \leftarrow \]
\[ \text{butter(filter.order, type = filt.type,} \]
\[ \text{W} = \text{pass.band}, \]
\[ \text{plane} = "s") \]
\[ \text{designed.filter} \leftarrow \]
\[ \text{butter(buttord(Wp = \text{pass.band}/\text{sample.rate}^2,} \]
\[ \text{Ws} = \text{stop.band}/\text{sample.rate}^2, \]
\[ \text{Rp} = 0.5, \text{Rs} = 40)) \# \text{ripple doesn't matter so much for} \]
\[ \text{# butter} \]

### 8.2.14 structures.R

```r
## initial data structure definitions
rbcie.env <- new.env() # environment for storing GUI state stuff
rbcie.env$opts <- list()
## list for datasets
## TODO rename
rbcie.env$importlist <- list()
## list for storing steps for later reporting
## hierarchy:
## -> summary.text string
## -> code.expr expression
## -> enabled boolean
rbcie.env$steplist <- list()
## example/test table:
# rbcie.env$steplist <-
# list(step1 =
# list(summary = "transform.pca of set.1",
# enabled = TRUE,
# code = "transform.pca(set.1,\nsome.params"),
# step2 =
# list(summary = "downsample.dt of set.2",
```

95
8.2.15  transformers.R

```r
# enabled = FALSE,
# code = "downsample.dt(set.2)"

libpci.env$pcaopts <- list()
library(kernlab) # nonlinear PCA

transform.pca <- function(targ.name = "Class",
                          epoch.name = "Trial",
                          time.name = "Time",
                          split.col = "Channel",
                          val.col = "Voltage",
                          input.table,
                          kernel.type,
                          pc.count = 2,
                          scale.data = FALSE,
                          pca.opts, # TODO needed?
                          pca.tol = sqrt(.Machine$double.eps)) {

  ## break out to channels for PCA function
  eeg.table <- channel.form(input.table,
                              value.col = val.col,
                              split.col = split.col,
                              class.col = targ.name,
                              time.col = time.name,
                              trial.col = epoch.name,
                              has.dups = FALSE)

  if (kernel.type == "Linear") {

    eeg.pca <- prcomp(reformulate(term.labels =
                               setdiff(colnames(eeg.table),
                                       c(targ.name,
                                         epoch.name,
                                         time.name))),
                      data = eeg.table,
                      scale = scale.data,
                      tol = pca.tol)

    # pull the pieces we need from each object into standard list?
    # for now, since it's not very big, let's just keep what we get
    return(eeg.pca)
  }

  else {
```
## nonlinear case

```r
switch(kernel.type,  
  Gaussian = {  
    # options: sigma  
    eeg.pca <- kpca(as.formula(paste("~",  
      "~",targ.name,  
      "~",epoch.name,  
      "~",time.name)),  
      data = eeg.table,  
      kernel = "rbfdot",  
      kpar = list(sigma = rbci.env$pcaopts$sigma),  
      features = pc.count, th = pca.tol)  
  },  
  Laplace = {  
    # options: sigma  
    eeg.pca <- kpca(as.formula(paste("~",  
      "~",targ.name,  
      "~",epoch.name,  
      "~",time.name)),  
      data = eeg.table,  
      kernel = "laplacedot",  
      kpar = list(sigma = rbci.env$pcaopts$sigma),  
      features = pc.count, th = pca.tol)  
  },  
  Polynomial = {  
    # options: degree, scale, offset  
    eeg.pca <- kpca(as.formula(paste("~",  
      "~",targ.name,  
      "~",epoch.name,  
      "~",time.name)),  
      data = eeg.table,  
      kernel = "polydot",  
      kpar = list(degree = rbci.env$pcaopts$degree,  
                   scale = rbci.env$pcaopts*scale,  
                   offset = rbci.env$pcaopts$offset),  
      features = pc.count, th = pca.tol)  
  },  
  Hyperbolic = {  
    # options: scale, offset  
    eeg.pca <- kpca(as.formula(paste("~",  
      "~",targ.name,  
      "~",epoch.name,  
      "~",time.name)),  
      data = eeg.table,  
      kernel = "tanhdot",  
      kpar = list(scale = rbci.env$pcaopts*scale,  
                   offset = rbci.env$pcaopts$offset),  
      features = pc.count, th = pca.tol)  
  }
```

97
Bessel = {
  # options: sigma, order, degree
  eeg.pca <- kPCA(formula(formula(paste("~ ",
                                         ",target.name,
                                         ",epoch.name,
                                         ",time.name)),
                        data = eeg.table,
                        kernel = "besseldot",
                        kpar = list(sigma = rbci.env$pcaopts$sigma,
                                     order = rbci.env$pcaopts$order,
                                     degree = rbci.env$pcaopts$degree),
                        features = pc.count, th = pca.tol)
  },
  ANOVA = {
    # options: sigma, degree
    eeg.pca <- kPCA(formula(formula(paste("~ ",
                                        ",target.name,
                                        ",epoch.name,
                                        ",time.name)),
                        data = eeg.table,
                        kernel = "laplacedot",
                        kpar = list(sigma = rbci.env$pcaopts$sigma,
                                     degree = rbci.env$pcaopts$degree),
                        features = pc.count, th = pca.tol)
  }
}
transform.pc <- function(pca.model,
                         target.name = "Class",
                         epoch.name = "Trial",
                         time.name = "Time",
                         split.col = "Channel",
                         value.col = "Voltage",
                         long.data.set) {
  ### TODO error checking on input

  ## convert long.data.set to wide form
  eeg.table <- channel.form(long.data.set,
                           value.col = val.col,
                           split.col = split.col,
                           class.col = target.name,
                           time.col = time.name,
                           trial.col = epoch.name,
                           has.dups = FALSE)
## do PCA prediction

```r
doPCA <- function(pca.model, newdata)
{
  pca.pred <- as.data.table(predict(pca.model, newdata = eeg.table))
  pca.pred[, targ.name := eeg.table[, get(targ.name)], with = FALSE]
  pca.pred[, epoch.name := eeg.table[, get(epoch.name)], with = FALSE]
  pca.pred[, targ.name := as.factor(get(targ.name)), with = FALSE]

  return(pca.pred)
}
```

### transform.kmeans

```r
transform.kmeans <- function(kmeans.data, val.col, kmeans.type, centers, max.iters) {
  kmeans(kmeans.data[, get(val.col)], centers, max.iters, algorithm = kmeans.type)
}
```

### transform.csp

```r
transform.csp <- function(table.data, time.col, chan.col, val.col, trial.col, class.col, avg.type = "Arithmetic", pair.count = 2)
{
  table.channel <- channel.form(table.data, value.col = val.col, split.col = chan.col, class.col = class.col, time.col = time.col, trial.col = trial.col, has.dups = FALSE)

  num.channels <- length(names(table.channel)) - 3
  if (pair.count == 0) { # all columns case
    pair.count <- num.channels/2
    ## TODO fit odd channels
  }

  ## construct vector pairing (first, last, etc.)
  pair.vec <- c((num.channels+1)-seq_len(pair.count), seq_len(pair.count))
}
```
## compute correlation matrices by trial, class

```
setkeyv(table.data,class.col)
setkeyv(table.channel,class.col) ### TODO review this
## build map of trials/classes
class.trial.map <- lapply(table.channel[,unique(get(class.col))],
                          function(x){
                          table.channel[J(x),unique(get(trial.col))]
                          })
correlation.mats.list <- # a list of lists; class( trial(...
lapply(class.trial.map, function(class.trials) {
  foreach (this.trial = class.trials,
           .combine = abind3curry) %do% {
    cor(table.channel[get(trial.col) == this.trial,
                   ### TODO fix these magic numbers (we're selecting the chan vars only)
                   names(table.channel)[4:length(names(table.channel))],
                   with=FALSE]
    })
  })
```

## str(correlation.mats.list)
## average matrices by class
avg.corr.mats <- lapply(correlation.mats.list, function(x) {
  ## each of these inputs is a cube of matrices
  ## we want to average over the 3rd dimension
  switch(avg.type,
         "Arithmetic" = {
        apply(x,c(1,2),mean) # just like that
         },
         "Geometric" = {
        apply(x,c(1,2),geometric.mean)
         },
         "Harmonic" = {
        apply(x,c(1,2),harmonic.mean)
         })
})

## jointly diagonalize
## make list of eigenvectors/eigenvalues, return
lapply(rjd(abind3curry(avg.corr.mats)), function(x) {
  ## return only as many pairs as required by input
  extract(x, indices=list("2"=pair.vec))
})
```

transform.cs <- function(csp.model,
```
targ.name,
epoch.name,
time.name,
split.col,
val.col,
long.data.set) {
### TODO error checking on input

#### converting from long table form to channel-split wide form ####
## for multivariate repeated time series

channel.form <- function(input.table,
value.col = "Voltage",
split.col = "Channel",
class.col = "Class",
time.col = "Time",
trial.col = "Trial",
has.dups = FALSE) {
  # TODO has.dups needed?
  # Converts long table format to slightly wider format split by channels.
  # For epoched datasets.
  if (has.dups == TRUE) {
    input.table[, c(trial.col):= sample.int(nrow(input.table))]
    setkeyv(input.table, trial.col)
    #trial.col = "id"
  }
  setkeyv(input.table, class.col)

  # magic begins here...
  chan.split <- split(input.table,input.table[, get(split.col)])
  chan.table <- cbind(lapply(chan.split,
    function(x){
      x[, value.col, with=FALSE]
    }))

  chan.table <-
  as.data.table(matrix(unlist(chan.table),
    ncol = input.table[, length(unique(get(split.col))))],
    byrow=TRUE))

  # reintroduce class labels
  # since the split is over identical sections for each channel, we can just
  # use the first split's labels
  chan.table <- chan.table[, c(class.col):= chan.split[[1]][, get(class.col)]]
  chan.table[, c(class.col):= as.factor(get(class.col))]

  # similarly with time and trial labels
chan.table <- chan.table[,Time:= chan.split[[1]][,get(time.col)]]
chan.table <- chan.table[,Trial:= chan.split[[1]][,get(trial.col)]]

setnames(chan.table,old=names(chan.table),
new=c(paste("Ch",input.table[,unique(get(split.col))],sep=""),
class.col,time.col,trial.col))

# data.table is not picky about column order, but we need to be in case of
# coercion to matrices etc.
setcolorder(chan.table,
class.col,time.col,trial.col,
paste("Ch",sort(input.table[,unique(get(split.col))]),sep="")))

return(chan.table)
}

### populate data set selector
if (!is.null(names(rbci.env$importlist))) {
  class_var_filesel <- gradio(names(rbci.env$importlist),
29 } else {
    class_var_filesel <- glabel("No data found.",
        container = class_var_frame)
}

35 class_task_book <- gnotebook(tab.pos = 3,
    container = class_pane)

38 sda_tab <- gframe(label = "SDA",
    container = class_task_book)

41 svm_tab <- gframe(label = "SVM",
    container = class_task_book)

44 bayes_tab <- gframe(label = "Naive Bayes",
    container = class_task_book)

48 source("./gwidgets/class_interface_sda.R")
49 source("./gwidgets/class_interface_svm.R")
50 source("./gwidgets/class_interface_bayes.R")
52 # set some widths (doesn't work if earlier)
53 svalue(class_pane) <- 0.3

8.2.17 class_interface_bayes.R

# button pane
bayes_pane <- gpanedgroup(horizontal = TRUE,
    expand = TRUE,
    fill = TRUE,
    container = bayes_tab)

bayes_param_group <- ggroup(container = bayes_pane,
    horizontal = TRUE)

bayes_varlist_frame <- gframe(text = "Feature Columns",
    container = bayes_param_group,
    expand = TRUE,
    width = 300)

## populate varlist
bayes_varlist <- gcheckboxgroup(  
    names(rbci.env$importlist[[svalue(class_var_filesel, index=TRUE)]]),
    container = bayes_varlist_frame,
use.table = TRUE, expand = TRUE)

## if we change datasets, update interface elements
addHandlerChanged(class_var_filesel, 
    handler = function(h,...) { 
        new.dataset.names <- 
            names(rbci.env$importlist[[svalue(class_var_filesel, 
                index=TRUE)]])
        bayes_varlist[] <- new.dataset.names
        bayes_target_list[] <- new.dataset.names
    })

## bayes params
bayes_param_frame <- gframe(text = "Naive Bayes Parameters", 
    horizontal = FALSE, 
    container = bayes_param_group, 
    expand = TRUE, 
    width = 300)

## opts

# numerical entries (spinboxes)
bayes_band_label <- glabel(text = "Numerical Parameters", 
    container = bayes_param_frame)
bayes_band_layout <- glayout(container = bayes_param_frame)

bayes_band_layout[1,1] <- "Laplace smoothing"
bayes_band_layout[2,1] <- gspinbutton(from = 0, by = 0.01)

## application params
bayes_target_frame <- gframe(text = "Target Variable", 
    horizontal = FALSE, 
    container = bayes_param_frame)

bayes_target_list <- 
gcombobox( 
    container = bayes_target_frame, 
    names(rbci.env$importlist[[svalue(class_var_filesel, index=TRUE)]]))

## output params
bayes_output_frame <- gframe(text = "Output Options", 
    horizontal = FALSE, 
    container = bayes_param_frame, 
    expand = TRUE, 
    width = 300)
bayes_output_layout <- glayout(container = bayes_output_frame)

## apply bayes button
bayes_output_layout[1,1] <-
  gbutton("Train Model",
         handler = function(h,...){
           train.name <- svalue(class_var_filesel)

           this.args <- list(# collect args
                              train.data = bquote( # partial deref
                                                        rbci.env$importlist[[.(train.name)]]),
                              bayes.smooth = svalue(bayes_band_layout[2,1]),
                              target.col = svalue(bayes_target_list),
                              feature.cols = svalue(bayes_varlist)
           )

           ## do the train model call
           new.table <- list(do.call(train.bayes.model,this.args))
           names(new.table) <- paste(train.name,
                                        "bayesmodel", seq_along(new.table),
                                        sep = ".")
           rbci.env$importlist <- append(rbci.env$importlist,
                                          new.table)

           ## ensure names are straight
           names(rbci.env$importlist) <-
           make.unique(names(rbci.env$importlist))

           ## update reporter with op
           add.step("train.bayes.model", this.args)
         })

## refresh dataset frame on run
## alert complete (progress bar?)

bayes_output_layout[1,2] <-
  gbutton("Print Table",
           handler = function(h,...){
           bayes.name <- svalue(class_var_filesel)
           data.name <- svalue(bayes_test_list)
           target.col <- svalue(bayes_target_list)

           this.args <- list(# collect args
                              # partial deref
                              bayes.prediction =
                              bquote(rbci.env$importlist[[.(bayes.name)]]),
                              test.data = bquote(c(
                                                     rbci.env$importlist[[.(data.name)]][,.](target.col),
                                                     rbci.env$importlist[[.(target.col)]]))

           new.table <- list(do.call(test.bayes.model,test.data),
                              do.call(predict.bayes.model,bayes.prediction))
           names(new.table) <- paste(bayes.name,
                                        "bayestable", seq_along(new.table),
                                        sep = ".")
           rbci.env$importlist <- append(rbci.env$importlist,
                                          new.table)

           ## ensure names are straight
           names(rbci.env$importlist) <-
           make.unique(names(rbci.env$importlist))

           ## update reporter with op
           add.step("test.bayes.model", this.args)
         })
## send table to widget
svalue(bayes_output_frame) <-
  ## we have to capture output here due to GUI
capture.output(do.call(table.bayes.model, this.args))

## send op to reporter
add.step("table.bayes.model", this.args)
}

bayes_output_layout[,3] <-
gbutton("Print Model",
  handler = function(h,...){
    bayes.name <- svalue(class_var_filesel)
    print.args <- list(
      x = bquote(
        rbci.env$importlist[[.(bayes.name)]]))
    
    svalue(bayes_output_frame) <-
      capture.output(
        do.call(print, print.args)
      )
    add.step("print", print.args)
  })

bayes_test_btn <-
gbutton("Test Model",
  container = bayes_output_frame,
  handler = function(h,...) {
    test.name <- svalue(class_var_filesel)
    test.dataname <- svalue(bayes_test_list)
    test.feats <- svalue(bayes_varlist)
    
    test.args <- list(  # collect partially dereferenced args
      bayes.model = bquote(
        rbci.env$importlist[[.(test.name)]]),
      test.data = bquote(
        rbci.env$importlist[[.(test.dataname)]]))
    
    ## do work, update GUI
    new.table <- list(  # do the model test with the above args
do.call(test.bayes.model, test.args)
)

names(new.table) <- paste(test.dataname, "bayestest", seq_along(new.table), sep = ".")
rbcie.env$importlist <- append(rbcie.env$importlist, new.table)

## ensure names are straight

names(rbcie.env$importlist) <- make.unique(names(rbcie.env$importlist))

## add op to reporter

add.step("test.bayes.model", test.args)
)

bayes_test_label <- glabel("Test Set", container = bayes_output_frame)
bayes_test_list <- gdroplist(container = bayes_output_frame, names(rbcie.env$importlist))

## we ALSO want to have column selector change when selecting target sets for
## the target set list

addHandlerChanged(bayes_test_list, handler = function(h, ...) {
  new.dataset.names <- names(rbcie.env$importlist[[svalue(bayes_test_list)]])

  if (!is.null(new.dataset.names)) {
    bayes_varlist[] <- new.dataset.names
    bayes_target_list[] <- new.dataset.names
  }
})

## Buttons that add new things should refresh the dataset selector

addHandlerClicked(bayes_output_layout[1,1],
  handler = function(h, ...){
    new.datasets <- names(rbcie.env$importlist)
    class_var_filesel[] <- new.datasets
  })

addHandlerClicked(bayes_test_btn,
  handler = function(h, ...){
    bayes_test_list[] <- names(rbcie.env$importlist)
  })
8.2.18  class_interface_sda.R

```r
## button pane
sda_pane <- gpanedgroup(horizontal = TRUE,
                         expand = TRUE,
                         fill = TRUE,
                         container = sda_tab)

sda_param_group <- ggroup(container = sda_pane,
                           horizontal = TRUE)

sda_varlist_frame <- gframe(text = "Feature Columns",
                            horizontal = TRUE,
                            container = sda_param_group,
                            expand = TRUE,
                            width = 300)

## populate varlist
sda_varlist <- gcheckboxgroup(
                names(rbci.env$importlist[[svalue(class_var_filesel, index=TRUE)]]),
                container = sda_varlist_frame,
                use.table = TRUE,
                expand = TRUE)

## if we change datasets, update interface elements
addHandlerChanged(class_var_filesel,
       handler = function(h,...) {
       new.dataset.names <-
               names(rbci.env$importlist[[svalue(class_var_filesel)]]))
       sda_varlist[] <- new.dataset.names
       sda_target_list[] <- new.dataset.names
       })
```

```r
## sda params
sda_param_frame <- gframe(text = "SDA Parameters",
                          horizontal = FALSE,
```
container = sda_param_group,
expand = TRUE,
width = 300)

## opts

# numerical entries (spinboxes)
sda_band_label <- glabel(text = "Numerical Parameters",
                         container = sda_param_frame)
sda_band_layout <- glayout(container = sda_param_frame)

sda_band_layout[1,1] <- "Shrinkage (correlation matrix)"
sda_band_layout[2,1] <- gspinbutton(from = 0, to = 1, by = 0.01)

# stop band end
sda_band_layout[3,1] <- "Shrinkage (variances)"
sda_band_layout[4,1] <- gspinbutton(from = 0, to = 1, by = 0.01)

## application params
sda_target_frame <- gframe(text = "Target Variable",
                           horizontal = FALSE,
                           container = sda_param_frame)

sda_target_list <-
gcombobox(
  container = sda_target_frame,
  names(rbci.env$importlist[[svalue(class_var_filesel, index=TRUE)]]))

## output params
sda_output_frame <- gframe(text = "Output Options",
                           horizontal = FALSE,
                           container = sda_param_frame,
                           width = 300)
sda_output_layout <- glayout(container = sda_output_frame)

# apply sda button
sda_output_layout[1,1] <-
gbutton("Train Model",
        handler = function(h,...){
          train.name <- svalue(class_var_filesel)

          train.args <- list(
            train.data = bquote( # partial deref of dataset call
                                  rbci.env$importlist[[.(train.name)]]),
            target.col = svalue(sda_target_list),
            feature.cols = svalue(sda_varlist),
            sda.lambda = svalue(sda_band_layout[2,1]),
          )
        })
sda.lambda.var = svalue(sda_band_layout[4,1])

new.table <- # do the train model call
  list(do.call(train.sda.model, train.args))
names(new.table) <- paste(train.name,
  "sdamodel", seq_along(new.table),
  sep = ".")
rbcie.env$importlist <- append(rbcie.env$importlist,
  new.table)

## ensure names are straight
names(rbcie.env$importlist) <- make.unique(names(rbcie.env$importlist))

## update reporter with op
add.step("train.sda.model", train.args)
}

## refresh dataset frame on run
## alert complete (progress bar?)

sda_output_layout[,1,2] <-
  gbutton("Print Table",
    handler = function(h,...) {
      sda.name <- svalue(class_var_filesel)
      data.name <- svalue(sda_test_list)
      target.col <- svalue(sda_target_list)

      table.args <- list( # collect table arguments
        sda.prediction = bquote( # partially deref data calls
          rbci.env$importlist[[.(sda.name)]],
          test.data =
          bquote(
            rbci.env$importlist[[.(data.name)]][,.(target.col),
            with=FALSE]
          )
      )

      ## send table to widget
      svalue(sda_output_frame) <-
        capture.output(do.call(table.sda.model, table.args))

      ## send op to reporter
      add.step("table.sda.model", this.args)
    })

sda_output_layout[,1,3] <-
  gbutton("Print Model",}
handler = function(h,...){
  sda.name <- svalue(class_var_filesel)
  print.args <- list(
    x = bquote(
      rbci.env$importlist[[.)(sda.name)]])
  )

  svalue(sda_output_frame) <-
  capture.output(
    do.call(print, print.args)
  )
  add.step("print", print.args)
}

sda_test_btn <-
  gbutton("Test Model",
    container = sda_output_frame,
    handler = function(h,...){
      test.name <- svalue(class_var_filesel)
      test.dataname <- svalue(sda_test_list)

      ## collect args
      test.args <- list(
        sda.model = bquote(
          rbci.env$importlist[[.)(test.name)]]),
        test.data = bquote(
          rbci.env$importlist[[.)(test.dataname)]]),
        feature.cols = svalue(sda_varlist)
      )

      ## do work, update GUI
      new.table <-
        list(do.call(test.sda.model, test.args))
      names(new.table) <- paste(test.dataname,
        "sdatest", seq_along(new.table),
        sep = ".")
      rbci.env$importlist <- append(rbci.env$importlist,
        new.table)

      ## ensure names are straight
      names(rbci.env$importlist) <-
        make.unique(names(rbci.env$importlist))

      ## add op to reporter
      add.step("test.sda.model", test.args)
    })
sd_test_label <- glabel("Test Set",
    container = sda_output_frame)

sd_test_list <-
gdroplist(container = sda_output_frame,
    names(rbci.env$importlist))

## we ALSO want to have column selector change when selecting target sets for
## the target set list
addHandlerChanged(sd_test_list,
    handler = function(h,...) {
      new.dataset.names <-
        names(rbci.env$importlist[[svalue(sd_test_list)])

      if (!is.null(new.dataset.names)) {
        sda_varlist[] <- new.dataset.names
        sda_target_list[] <- new.dataset.names
      }
    })

## Buttons that add new things should refresh the dataset selector
addHandlerClicked(sda_output_layout[,1],
    handler = function(h,...){
      new.datasets <-
        names(rbci.env$importlist)
      class_var_filename[] <- new.datasets
      sda_test_list[] <- new.datasets
    })

addHandlerClicked(sd_test_btn,
    handler = function(h,...){
      new.datasets <-
        names(rbci.env$importlist)
      sda_test_list[] <- new.datasets
    })

sd_output_frame <- gtext(text = "sda output",
    font.attr=c(family="monospace"),
    # width = window.width*0.4,
    container = sda_pane)

# set some widths (doesn't work if earlier)
svalue(sda_pane) <- 0.6

8.2.19 class_interface_svm.R
## button pane

```r
svm_pane <- gpanedgroup(horizontal = TRUE,
                        expand = TRUE,
                        fill = TRUE,
                        container = svm_tab)
```

```r
svm_param_group <- ggroup(container = svm_pane,
                        horizontal = TRUE)
```

```r
svm_varlist_frame <- gframe(text = "Feature Columns",
                        horizontal = FALSE,
                        container = svm_param_group,
                        expand = TRUE,
                        full = TRUE,
                        width = 300)
```

## populate varlist

```r
svm_varlist <- gcheckboxgroup(
    names(rbci.env$importlist[[svalue(class_var_filesel, index=TRUE)]]),
    container = svm_varlist_frame,
    use.table = TRUE,
    expand = TRUE)
```

## if we change datasets, update interface elements

```r
addHandlerChanged(class_var_filesel,
    handler = function(h,...) {
        new.dataset.names <-
        names(rbci.env$importlist[[svalue(class_var_filesel, index=TRUE)]])
        svm_varlist[] <- new.dataset.names
        svm_target_list[] <- new.dataset.names
    })
```

## svm params

```r
svm_param_frame <- gframe(text = "SVM Parameters",
                        horizontal = FALSE,
                        container = svm_param_group,
                        expand = TRUE,
                        width = 300)
```

## opts

```r
svm_kernel_type_list <- c("Linear", "Gaussian",
                        "Laplace", "Polynomial",
                        "Hyperbolic", "Bessel",
                        "ANOVA RBF", "Spline")
svm_kernel_type_label <- glabel(text = "Kernel Type (library)",
...)
```
svm_kernel_type_menu <-
  gdroplist(svm_kernel_type_list,
    text = "Kernel Type",
    container = svm_param_frame,
    handler = function (h,...) {
      # enable or disable param GUI opts on type change

      ## param names by type
      # sigma for rbf/Laplace
      # degree, scale, offset for Polynomial
      # scale, offset for tanhdot
      # sigma, order, degree, for Bessel
      # sigma, degree for ANOVA
      switch (svalue(h$obj),
        "Linear" = {
          enabled(svm_band_layout[1,1]) <- FALSE
          enabled(svm_band_layout[2,1]) <- FALSE
          enabled(svm_band_layout[1,2]) <- FALSE
          enabled(svm_band_layout[2,2]) <- FALSE
          enabled(svm_band_layout[3,1]) <- FALSE
          enabled(svm_band_layout[4,1]) <- FALSE
        },
        "Laplace" = {
          enabled(svm_band_layout[1,1]) <- TRUE
          enabled(svm_band_layout[2,1]) <- TRUE
          svalue(svm_band_layout[1,1]) <- "Sigma"
          enabled(svm_band_layout[1,2]) <- FALSE
          enabled(svm_band_layout[2,2]) <- FALSE
          enabled(svm_band_layout[3,1]) <- FALSE
          enabled(svm_band_layout[4,1]) <- FALSE
        },
        "Gaussian" = {
          enabled(svm_band_layout[1,1]) <- TRUE
          enabled(svm_band_layout[2,1]) <- TRUE
          svalue(svm_band_layout[1,1]) <- "Sigma"
          enabled(svm_band_layout[1,2]) <- FALSE
          enabled(svm_band_layout[2,2]) <- FALSE
          enabled(svm_band_layout[3,1]) <- FALSE
          enabled(svm_band_layout[4,1]) <- FALSE
        },
        "Polynomial" = {
          enabled(svm_band_layout[1,1]) <- TRUE
          enabled(svm_band_layout[2,1]) <- TRUE
          svalue(svm_band_layout[1,1]) <- "Degree"
enabled(svm_band_layout[1,2]) <- TRUE
enabled(svm_band_layout[2,2]) <- TRUE
svalue(svm_band_layout[1,2]) <- "Scale"

enabled(svm_band_layout[3,1]) <- TRUE
enabled(svm_band_layout[4,1]) <- TRUE
svalue(svm_band_layout[3,1]) <- "Offset"

"Hyperbolic" = {
  enabled(svm_band_layout[1,1]) <- TRUE
  enabled(svm_band_layout[2,1]) <- TRUE
  svalue(svm_band_layout[1,1]) <- "Scale"

  enabled(svm_band_layout[1,2]) <- TRUE
  enabled(svm_band_layout[2,2]) <- TRUE
  svalue(svm_band_layout[1,2]) <- "Offset"

  enabled(svm_band_layout[3,1]) <- FALSE
  enabled(svm_band_layout[4,1]) <- FALSE
},

"Bessel" = {
  enabled(svm_band_layout[1,1]) <- TRUE
  enabled(svm_band_layout[2,1]) <- TRUE
  svalue(svm_band_layout[1,1]) <- "Sigma"

  enabled(svm_band_layout[1,2]) <- TRUE
  enabled(svm_band_layout[2,2]) <- TRUE
  svalue(svm_band_layout[1,2]) <- "Order"

  enabled(svm_band_layout[3,1]) <- TRUE
  enabled(svm_band_layout[4,1]) <- TRUE
  svalue(svm_band_layout[3,1]) <- "Degree"
},

"ANOVA RBF" = {
  enabled(svm_band_layout[1,1]) <- TRUE
  enabled(svm_band_layout[2,1]) <- TRUE
  svalue(svm_band_layout[1,1]) <- "Sigma"

  enabled(svm_band_layout[1,2]) <- TRUE
  enabled(svm_band_layout[2,2]) <- TRUE
  svalue(svm_band_layout[1,2]) <- "Degree"

  enabled(svm_band_layout[3,1]) <- FALSE
  enabled(svm_band_layout[4,1]) <- FALSE
},

"Spline" = {
  enabled(svm_band_layout[1,1]) <- FALSE
  enabled(svm_band_layout[2,1]) <- FALSE

  enabled(svm_band_layout[3,1]) <- FALSE
  enabled(svm_band_layout[4,1]) <- FALSE
},
enabled(svm_band_layout[1,2]) <- FALSE
enabled(svm_band_layout[2,2]) <- FALSE

enabled(svm_band_layout[3,1]) <- FALSE
enabled(svm_band_layout[4,1]) <- FALSE
}

## target variable
svm_target_label <- glabel("Target Variable",
    container = svm_param_frame)

svm_target_list <-
gcombobox(
    container = svm_param_frame,
    names(rbci.env$importlist[[svalue(class_var_filesel, index=TRUE)]]))

## numerical entries (spinboxes)
svm_band_label <- glabel(text = "Numerical Parameters",
    container = svm_param_frame)

svm_band_layout <- glayout(container = svm_param_frame)

svm_band_layout[1,1] <- ""
svm_band_layout[2,1] <- gspinbutton(from = 0, to = 1, by = 0.01)

svm_band_layout[1,2] <- ""
svm_band_layout[2,2] <- gspinbutton(from = 0, to = 1, by = 0.01)

svm_band_layout[3,1] <- ""
svm_band_layout[4,1] <- gspinbutton(from = 0, to = 1, by = 0.01)

svm_band_layout[3,2] <- "Cost"
svm_band_layout[4,2] <- gspinbutton(from = 0, to = length(svm_varlist), by = 1)

enabled(svm_band_layout[1,1]) <- FALSE
enabled(svm_band_layout[2,1]) <- FALSE
enabled(svm_band_layout[1,2]) <- FALSE
enabled(svm_band_layout[2,2]) <- FALSE
enabled(svm_band_layout[3,1]) <- FALSE
enabled(svm_band_layout[4,1]) <- FALSE

## application params
# training/test fractions

## output params
svm_output_frame <- gframe(text = "Output Options",
    horizontal = FALSE,
container = svm_param_frame,
expand = TRUE)

svm_output_layout <- glayout(container = svm_output_frame)

# apply svm button
svm_output_layout[1,1] <-
gbutton("Train Model",
handler = function(h,...){
  train.name <- svalue(class_var_filesel)

  ## collect args
  train.args <- list(
    train.data = bquote(# partially deref dataset calls
                         rbci.env$importlist[[.(train.name)]]),
    kern.type = svalue(svm_kernel_type_menu),
    target.col = svalue(svm_target_list),
    feature.cols = svalue(svm_varlist),
    cost.param = svalue(svm_band_layout[4,2]),
    kern.params = list(
      svalue(svm_band_layout[2,1]),
      svalue(svm_band_layout[2,2]),
      svalue(svm_band_layout[4,1]))
  )

  ## do the call, update GUI
  new.table <- list(do.call(train.svm.model, train.args))
  names(new.table) <- paste(train.name,
                            "svmmodel", seq_along(new.table),
                            sep = ".")
  rbci.env$importlist <- append(rbci.env$importlist, new.table)

  ## ensure names are straight
  names(rbci.env$importlist) <-
  make.unique(names(rbci.env$importlist))

  ## update reporter with op
  add.step("train.svm.model", train.args)
})

svm_test_btn <-
gbutton("Test Model",
container = svm_output_frame,
handler = function(h,...){
  svm.name <- svalue(class_var_filesel)
  test.dataname <- svalue(svm_test_list)

  ## collect args
test.args <- list(
  test.data = bquote( # partially dereference dataset calls
    rbci.env$importlist[[.(test.dataname)]]),
  svm.model = bquote(
    rbci.env$importlist[[.(svm.name)]]),
  feature.cols = svalue(svm_varlist)
)

## do the work, update the GUI
new.table <- list(do.call(test.svm.model, test.args))
names(new.table) <- paste(test.dataname,
                           "svmtest", seq_along(new.table),
                           sep = ".")
rbcirenv$importlist <- append(rbcirenv$importlist,
                               new.table)

## ensure names are straight
names(rbcirenv$importlist) <- make.unique(names(rbcirenv$importlist))

## update reporter with op
add.step("test.svm.model", test.args)
)

svm_test.label <-
  glabel("Test Set",
          container = svm_output_frame)

svm_test_list <-
  gdroplist(container = svm_output_frame,
            names(rbcirenv$importlist))

## we ALSO want to have column selector change when selecting target sets for
## the target set list
addHandlerChanged(svm_test_list,
                   handler = function(h,...) {
                     new.dataset.names <-
                       names(rbcirenv$importlist[[svalue(svm_test_list)]]))
                     if (!is.null(new.dataset.names)) {
                       svm_varlist[] <- new.dataset.names
                       svm_target_list[] <- new.dataset.names
                     }
                   })

svm_output_layout[1,2] <-
  gbutton("Print Table",
          handler = function(h,...) {

svm.name <- svalue(class_var_filesel)
data.name <- svalue(svm_test_list)
target.col <- svalue(svm_target_list)

## collect args
table.args <- list(
  svm.prediction = bquote(# partially deref dataset calls
    rbci.env$importlist[[.(svm.name)]]),
  test.data = bquote(
    rbci.env$importlist[[.(data.name)][,.(target.col),
    with=FALSE])
)
browser()
## send table to widget
svalue(svm_output_frame) <-
capture.output(do.call(table.svm.model, table.args))

## update reporter with op
add.step("table.svm.model", table.args)
}

svm_output_layout[1,3] <-
gbutton("Print Model",
  handler = function(h,...) {
    svm.name <- svalue(class_var_filesel)

    print.args <- list(
      x = bquote(
        rbci.env$importlist[[.(svm.name)]])
    )

    svalue(svm_output_frame) <-
capture.output(
      do.call(print, print.args)
    )
    add.step("print", print.args)
  })

## Buttons that add new things should refresh the dataset selector
addHandlerClicked(svm_output_layout[1,1],
  handler = function(h,...){
    new.datasets <-
      names(rbci.env$importlist)
    class_var_filesel[] <- new.datasets
  })
addHandlerClicked(svm_test_btn,
  handler = function(h,...){
    new.datasets <-
```r
names(rbci.env$importlist)
class_var_filesel[] <- new.datasets
}

## plot pane
svm_output_frame <- gtext(text = "svm output",
font.attr = c(family = "monospace"),
#
width = window.width*0.4,
container = svm_pane)

# set some widths (doesn't work if earlier)
svalue(svm_pane) <- 0.5

8.2.20  explore_interface.R

window.width <- 1000
window.height <- 600
preview.rowlen <- 20

explore_win <- gwindow("Data Explorer",
width = window.width,
height = window.height)

explorePane <- gpanedgroup(horizontal = TRUE,
expand = TRUE,
fill = TRUE,
container = explore_win)

explore_var_group <- ggroup(use.scrollwindow = TRUE,
horizontal = FALSE,
expand = TRUE,
container = explorePane,
width = 200)

explore_var_frame <- gframe(text = "Data Sets",
horizontal = FALSE,
container = explore_var_group,
expand = TRUE)

## populate data set selector
if (!is.null(names(rbci.env$importlist))) {
    explore_var_filesel <- gradio(names(rbci.env$importlist),
    container = explore_var_frame)
} else {
```
explore_var_filesel <- glabel("No data found.",
        container = explore_var_frame)
}

explore_task_book <- gnotebook(tab.pos = 3,
        container = explorePane)

summary_tab <- gframe(label = "Summary",
        container = explore_task_book)

means_tab <- gframe(label = "Grand Means",
        container = explore_task_book)

hist_tab <- gframe(label = "Histogram",
        container = explore_task_book)

partition_tab <- gframe(label = "Partition",
        container = explore_task_book)

# Load subitems (into tabs)
source("./gwidgets/explore_interface_summary.R")
source("./gwidgets/explore_interface_means.R")
source("./gwidgets/explore_interface_hist.R")
source("./gwidgets/explore_interface_partition.R")

# re-init backend (TODO re-organize this)
source("./backend/explorers.R")
source("./backend/partition_table.R")

# set some widths (doesn’t work if earlier)
svalue(explorePane) <- 0.2

8.2.21 explore_interface_annotate.R

# only changes variable types for now (remember to document)
annotate_mainframe <- gpanedgroup(horizontal = TRUE,
        expand = TRUE,
        fill = TRUE,
        container = annotateTab)

annotate_varlist_frame <- gframe(text = "Data Columns",
        horizontal = FALSE,
        container = annotate_mainframe,
```r
### populate varlist
annotate_varlist <- gradio(
  names(rbci.env$importlist[[svalue(explore_var_filesel, index=TRUE)]]),
  container = annotate_varlist_frame,
  ## use.table = TRUE,
  expand = FALSE,
  handler = function(h, ...) {
    ## update current datatype
    svalue(annotate_curvartype) <-
      paste("Current Data Type:",
            sapply(rbci.env$importlist[[svalue(explore_var_filesel)]],
                   class)[svalue(annotate_varlist)]

    ## TODO: are these updates needed? (probably not)
    ## update current tag
    svalue(annotate_curtarget) <-
      paste("Current Target Column:",
            rbci.env$tags[[svalue(explore_var_filesel)]]$targetcol)

    ## update current tag
    svalue(annotate_curvalue) <-
      paste("Current Value Column:",
            rbci.env$tags[[svalue(explore_var_filesel)]]$valuecol)

    svalue(annotate_curepoch) <-
      paste("Current Epoch Column:",
            rbci.env$tags[[svalue(explore_var_filesel)]]$epochcol)
  }

addSpring(annotate_varlist_frame)

### TODO second widget group for creating additional columns
annotate_optframe <- gframe(text = "Annotation Options",
                             container = annotate_mainframe,
                             horizontal = FALSE)

## variable type
glabel("Data Type",
       container = annotate_optframe,
       anchor = c(-1,1))

annotate_curvartype <- glabel("Current Data Type: ",
                               container = annotate_optframe,
                               anchor = c(-1,1))
```
glabell("New Data Type",
    container = annotate_optframe,
    anchor = c(-1,1))

annotate_datatype_chooser <-
gdroplist(c("Numeric","Integer","Complex","Logical","Character"),
    container = annotate_optframe,
    anchor = c(-1,1))

annotate_datatype_apply <-
gbutton("Apply",
    container = annotate_optframe,
    handler = function(h, ...) {
        curfile <- svalue(explore_var_filesel)
        curcol <- svalue(annotate_varlist)
        switch(svalue(annotate_datatype_chooser),
            Numeric = {
                rbci.env$importlist[[curfile]][, curcol :=
                as.numeric(get(curcol))]
                },
            Integer = {
                rbci.env$importlist[[curfile]][, curcol :=
                as.integer(get(curcol))]
                },
            Complex = {
                rbci.env$importlist[[curfile]][, curcol :=
                as.complex(get(curcol))]
                },
            Logical = {
                rbci.env$importlist[[curfile]][, curcol :=
                as.logical(get(curcol))]
                },
            Character = {
                rbci.env$importlist[[curfile]][, curcol :=
                as.character(get(curcol))]
                })
        svalue(annotate_curvartype) <-
paste("Current Data Type:",
        sapply(rbci.env$importlist[[curfile]],
        class)[curcol])
    })

## interest tag: target, value
glabell("Column Tag",
    container = annotate_optframe,
    anchor = c(-1,1))
annotate_datatype_chooser <-
gdroplist(c("No Tag", "Target Column", "Value Column", "Epoch Column", 
"Time Column", "Channel Column"),
  container = annotate_optframe,
  anchor = c(-1,1),
  handler = function(h, ...) {
    curfile <- svalue(explore_var_filesel)
    curcol <- svalue(annotate_varlist)

    ## TODO clean up this vile switch statement
    if ( svalue(h$obj) == "Target Column") {
      ## set target tag
      rbci.env$tags[[curfile]]$targetcol <- curcol
      ## update display
      svalue(annotate_curtarget) <-
        paste("Current Target Column:", curcol)
    } else if ( svalue(h$obj) == "Value Column") {
      ## set value tag
      rbci.env$tags[[curfile]]$valuecol <- curcol
      ## update display
      svalue(annotate_curvalue) <-
        paste("Current Value Column:", curcol)
    } else if ( svalue(h$obj) == "Epoch Column") {
      ## set value tag
      rbci.env$tags[[curfile]]$epochcol <- curcol
      ## update display
      svalue(annotate_curepoch) <-
        paste("Current Epoch Column:", curcol)
    } else if ( svalue(h$obj) == "Time Column") {
      ## set value tag
      rbci.env$tags[[curfile]]$timecol <- curcol
      ## update display
      svalue(annotate_curtime) <-
        paste("Current Time Column:", curcol)
    } else if ( svalue(h$obj) == "Channel Column") {
      ## set value tag
      rbci.env$tags[[curfile]]$chancol <- curcol
      ## update display
      svalue(annotate_curchan) <-
        paste("Current Channel Column:", curcol)
    } else if ( svalue(h$obj) == "No Tag") {
      ## match tags, then delete if needed
      if (svalue(annotate_curtarget) ==

124
```r
paste("Current Target Column:", curcol)) {
  rbci.env$tags[[curfile]]$targetcol <- NULL

  svalue(annotate_curtarget) <-
  paste("Current Target Column: None")
}
if (svalue(annotate_curvalue) <-
paste("Current Value Column:", curcol)) {
  rbci.env$tags[[curfile]]$valuecol <- NULL

  svalue(annotate_curvalue) <-
  paste("Current Value Column: None")
}
if (svalue(annotate_curepoch) <-
paste("Current Epoch Column:", curcol)) {
  rbci.env$tags[[curfile]]$epochcol <- NULL

  svalue(annotate_curepoch) <-
  paste("Current Epoch Column: None")
}
if (svalue(annotate_curtime) <-
paste("Current Time Column:", curcol)) {
  rbci.env$tags[[curfile]]$timecol <- NULL

  svalue(annotate_curtime) <-
  paste("Current Time Column: None")
}
if (svalue(annotate_curchan) <-
paste("Current Channel Column:", curcol)) {
  rbci.env$tags[[curfile]]$chancol <- NULL

  svalue(annotate_curchan) <-
  paste("Current Channel Column: None")
}
}
}
annotate_curtarget <-
  glabel("Current Target Column:",
            container = annotate_optframe,
            anchor = c(-1,1))
annotate_curvalue <-
  glabel("Current Value Column:",
            container = annotate_optframe,
```

anchor = c(-1,1))

annotate_curepoch <-
glabel("Current Epoch Column:",
       container = annotate_optframe,
       anchor = c(-1,1))

annotate_curtime <-
glabel("Current Time Column:",
       container = annotate_optframe,
       anchor = c(-1,1))

annotate_curchan <-
glabel("Current Channel Column:",
       container = annotate_optframe,
       anchor = c(-1,1))

svalue(annotate_mainframe) <- 0.25

8.2.22  explore_interface_hist.R

hist_pane <- gpanedgroup(horizontal = TRUE,
                         expand = TRUE,
                         fill = TRUE,
                         container = hist_tab)

hist_varlist_frame <- gframe(text = "Data Columns",
                         horizontal = FALSE,
                         container = hist_pane,
                         expand = TRUE,
                         width = 300)

# populate varlist
hist_varlist <- gcheckboxgroup(
    names(rbci.env$importlist[[svalue(explore_var_filesel, index=TRUE)]]),
    container = hist_varlist_frame,
    use.table = TRUE,
    expand = TRUE)

hist_summarize_btn <-
gbutton(text = "Plot Histogram",
        container = hist_varlist_frame,
        handler = function(h,...) {
            this.args <- list(
                data.set =
                bquote( # partially dereference call

126
26       rbc.env$importlist[[.(svalue(explore_var_filesel)]][],
27             .(svalue(hist_varlist)),
28             with = FALSE )
29          )
30
31          visible(hist_output_frame) <- TRUE
32          print(do.call(hist.plot,this.args)) # do the GUI Work
33
34          ## update reporter module with this op
35          add.step(func.name = "hist.plot",
36                      step.args = this.args)
37          )}
38
39          hist_output_frame <- ggraphics(container = hist-pane)
40
41          svalue(hist-pane) <- 0.25

8.2.23  explore_interface_means.R

1          means-pane <- gpanedgroup(horizontal = TRUE,
2              expand = TRUE,
3              fill = TRUE,
4              container = means-tab)
5
6
7
8          means-param-frame <- gframe(text = "",
9              horizontal = FALSE,
10             container = means-pane,
11             expand = TRUE,
12             width = 300)
13          ## application params
14          means-grouping-frame <- gframe(text = "",
15              horizontal = FALSE,
16             container = means-param-frame,
17             expand = TRUE,
18             width = 300)
19
20 # trial/group vars
21      means-grouping-label <- glabel(text = "Data Grouping",
22             container = means-grouping-frame)
23      means-grouping-layout <- glayout(container = means-grouping-frame)
24
25      means-grouping-layout[1,1] <- "Data Variable (Voltage)"
26      means-grouping-layout[2,1] <-
gcombobox(
  names(rbci.env$importlist[[svalue(explore_var_filesel, index=TRUE)]]))

means_grouping_layout[3,1] <- "Time Variable (Sample)"
means_grouping_layout[4,1] <-
gcombobox(
  names(rbci.env$importlist[[svalue(explore_var_filesel, index=TRUE)]]))

means_grouping_layout[5,1] <- "Target Variable (Class)"
means_grouping_layout[6,1] <-
gcombobox(
  names(rbci.env$importlist[[svalue(explore_var_filesel, index=TRUE)]]))

means_grouping_layout[7,1] <- "Second Group (Channel)"
means_grouping_layout[8,1] <-
gcombobox(
  names(rbci.env$importlist[[svalue(explore_var_filesel, index=TRUE)]]))

## user alert if not present
means_plot_btn <-
gbutton(text = "Plot",
    container = means_param_frame,
    handler = function(h,...) {
      this.args <-
        list(
          eeg.table =
            bquote( # partially dereference call
              rbci.env$importlist[[.(svalue(explore_var_filesel))]]),
          val.name = svalue(means_grouping_layout[2,1]),
          time.name = svalue(means_grouping_layout[4,1]),
          chan.name = svalue(means_grouping_layout[8,1]),
          targ.name = svalue(means_grouping_layout[6,1])
        )

      print(do.call(
        grand.means.plot,
        this.args)
      )

      add.step(func.name = "grand.means.plot",
               step.args = this.args)
    })

means_output_frame <- ggraphics(container = means_pane)
svalue(means_pane) <- 0.2
8.2.24 explore_interface_partition.R

```r
## partition param
partition_param_frame <- gframe(text = "Partition Parameters",
                                 horizontal = FALSE,
                                 container = partition_tab,
                                 expand = TRUE)

### type
partition_type_list <- c("Random", "Sequential")
poll_type_label <- glabel(text = "Partition Type",
                           container = partition_param_frame)
poll_type_menu <-
    gdroplist(partition_type_list, 
              text = "Partition Type",
              container = partition_param_frame)

### numerical entry (spinboxes)
poll_band_label <- glabel(text = "Numerical Parameters",
                          container = partition_param_frame)
poll_band_layout <- glayout(container = partition_param_frame)
poll_band_layout[1,1] <- "Number of Sets"
poll_band_layout[2,1] <-
    gspinbutton(from = 2, to = 4, by = 1,
                value = 2, 
                handler = function(h,...){
                    switch(as.character(svalue(h$obj)),
                           "2"="{
                           enabled(partition_band_layout[2,2]) <- TRUE
                           enabled(partition_band_layout[3,1]) <- FALSE
                           enabled(partition_band_layout[4,1]) <- FALSE
                           enabled(partition_band_layout[3,2]) <- FALSE
                           enabled(partition_band_layout[4,2]) <- FALSE
                           }
                           "3"="{
                           enabled(partition_band_layout[2,2]) <- TRUE
                           enabled(partition_band_layout[3,1]) <- TRUE
                           enabled(partition_band_layout[4,1]) <- TRUE
                           enabled(partition_band_layout[3,2]) <- FALSE
                           enabled(partition_band_layout[4,2]) <- FALSE
                           },
                           "4"="{
                           enabled(partition_band_layout[1,2]) <- TRUE
                           }"
```

129
```r
enabled(partition_band_layout[2,2]) <- TRUE
enabled(partition_band_layout[3,1]) <- TRUE
enabled(partition_band_layout[4,1]) <- TRUE
enabled(partition_band_layout[3,2]) <- TRUE
enabled(partition_band_layout[4,2]) <- TRUE
}

partition_band_layout[1,2] <- "Set 1 Proportion"
partition_band_layout[2,2] <- gspinbutton(from = 0, to = 1, by = 0.001,
  value = 0.5)

partition_band_layout[3,1] <- "Set 2 Proportion"
partition_band_layout[4,1] <- gspinbutton(from = 0, to = 1, by = 0.001,
  value = 0.5)

partition_band_layout[3,2] <- "Set 3 Proportion"
partition_band_layout[4,2] <- gspinbutton(from = 0, to = 1, by = 0.001)

enabled.list(state = FALSE,
  partition_band_layout[3,1],
  partition_band_layout[3,2],
  partition_band_layout[4,1],
  partition_band_layout[4,2])

## application params
partition_grouping_frame <- gframe(text = "Partition Apply Rules",
  horizontal = FALSE,
  container = FALSE,
  expand = TRUE,
  width = 300)

## trial/group vars
partition_grouping_label <- glabel(text = "Data Grouping",
  container = FALSE)
partition_grouping_layout <- glayout(container = FALSE)

partition_grouping_layout[1,1] <- "Partition Group (Trials)"
partition_grouping_layout[2,1] <-
  gcombobox(
    names(rbci.env$importlist[[svalue(explore_var_filesel, index=TRUE)]])
  )

addHandlerChanged(explore_var_filesel, handler = function(h,...) {
  new.dataset.names <-
    names(rbci.env$importlist[[svalue(explore_var_filesel, index=TRUE)]])
  partition_grouping_layout[2,1][] <- new.dataset.names
})
```
## output params

partition_output_frame <- gframe(text = "Output Controls",
    horizontal = FALSE,
    container = partition_param_frame,
    expand = FALSE,
    width = 320)

# apply partition button

partition_apply_btn <-
gbutton("Apply Partition to Data",
    container = partition_output_frame,
    handler = function(h,...){
      ## proportion inputs depend on how many subsets we have
      switch(as.character(svalue(partition_band_layout[2,1])),
        "2"={
          part.props <-
            c(svalue(partition_band_layout[2,2]),
              1-svalue(partition_band_layout[2,2]))
        },
        "3"={
          prop.total <- sum(svalue(partition_band_layout[2,2]),
            svalue(partition_band_layout[4,1]))
          part.props <-
            c(svalue(partition_band_layout[2,2]),
              svalue(partition_band_layout[4,1]),
              1-prop.total)
        },
        "4"={
          ## normalize
          prop.total <- sum(svalue(partition_band_layout[2,2]),
            svalue(partition_band_layout[4,1]),
            svalue(partition_band_layout[4,2]))
          part.props <- c(svalue(partition_band_layout[2,2]),
            svalue(partition_band_layout[4,1]),
            svalue(partition_band_layout[4,2]),
            1-prop.total)
        })
      }

      file.name <- svalue(explore_var_filesel)

      this.args <-
      list(
        table.data = bquote( # partially deref call
          rbci.env$importlist[[.file.name]]),
        part.col = svalue(partition_grouping_layout[2,1]),
        )
proportions = part.props,
part.type = svalue(partition_type_menu)
)

## apply the partition, add new data tables to list
new.tables <- do.call(partition.table, this.args)

## naming for clarity
names(new.tables) <- paste(file.name,
    "part", seq_along(new.tables),
    sep = ".")

## update reporter with this op
add.step("partition.table", this.args)

addHandlerClicked(partition_apply_btn,
    handler = function(h,...){
    new.datasets <-
    names(rbci.env$importlist[[x]] <- new.tables[[x]]

    ## update reporter with this op
    add.step("partition.table", this.args)
})

# populate varlist
summary_varlist <- gcheckboxgroup(
    names(rbci.env$importlist[[svalue(explore_var_filesel, index=TRUE)]]),
    container = summary_varlist_frame,
    width = 300)

# populate varlist
summary_varlist <- gcheckboxgroup(
    names(rbci.env$importlist[[svalue(explore_var_filesel, index=TRUE)]])
    container = summary_varlist_frame,
use.table = TRUE,
expand = TRUE)

summary_summarize_btn <-
gbutton(text = "Summarize",
        container = summary_varlist_frame,
        handler = function(h,...) {
            this.args <-
                list(eeg.table = # partially dereference call
                     bquote(
                         rbci.env$importlist[[.(svalue(explore_var_filesel))]],
                         selected.columns = svalue(summary_varlist))
                svalue(summary_output_frame) <- # do the job
                do.call(summarize, this.args)

                ## update the reporter module with a record of this op
                add.step(func.name = "summarize",
                         step.args = this.args)
        })

summary_output_frame <- gtext(text = "Summary output",
                               font.attr=c(family="monospace"),
                               width = window.width*0.4,
                               container = summary_pane)

svalue(summary_pane) <- 0.25

8.2.26  filter_interface.R

window.width <- 1000
window.height <- 600

preview.rowlen <- 20

filter_win <- gwindow("Filter Data",
                       width = window.width,
                       height = window.height)

filter_pane <- gpanedgroup(horizontal = TRUE,
                            expand = TRUE,
                            fill = TRUE,
                            container = filter_win)

filter_var_group <- ggroup(use.scrollwindow = TRUE,
# populate data set selector
if (!is.null(names(rbci.env$importlist))) {
  filter_var_filesel <- gradio(names(rbci.env$importlist),
    container = filter_var_frame)
} else {
  filter_var_filesel <- glabel("No data found.",
    container = filter_var_frame)
}

filter_task_book <- gnotebook(tab.pos = 3,
  container = filter_pane)

filter_simple_tab <- gframe(label = "Simple Filtering",
  container = filter_task_book)

filter_downsample_tab <- gframe(label = "Downsampling",
  container = filter_task_book)

source("./gwidgets/filter_interface_simple.R")
source("./gwidgets/filter_interface_downsample.R")

# set some widths (doesn’t work if earlier)
svalue(filter_pane) <- 0.2

8.2.27 filter_interface_downsample.R


downsample_varlist_frame <- gframe(text = "Data Columns",
  horizontal = FALSE,
  container = filter_downsample_tab,
  expand = TRUE,
  width = 300)

# populate varlist
downsample_varlist <- gcheckboxgroup(
  names(rbci.env$importlist[[svalue(filter_var_filesel, index=TRUE)]]),
  container = downsample_varlist_frame,
  use.table = TRUE,
  expand = TRUE)
addHandlerChanged(filter_var_filesel,
    handler = function(h,...) {
        new.dataset.names <-
            names(rbci.env$importlist[[svalue(filter_var_filesel, index=TRUE)]]))
        downsample_varlist[] <- new.dataset.names
    })

## downsample params

downsample_param_frame <- gframe(text = "Downsample Parameters",
    horizontal = FALSE,
    container = filter_downsample_tab,
    expand = TRUE,
    width = 300)

### type

## downsample_type_list <- c("Lowpass","Bandpass","Highpass","Stopband")
## downsample_type_label <- glabel(text = "Downsample Type",
##    container = downsample_param_frame)
## downsample_type_menu <-
##    gdroplist(downsample_type_list,
##        text = "Downsample Type",
##        container = downsample_param_frame,
##        handler = function (h,...) {
##            # enable or disable band param GUI opts on type change
##        })

### numerical entry (spinboxes)

downsample_band_label <- glabel(text = "Numerical Parameters",
    container = downsample_param_frame)
downsample_band_layout <- glayout(container = downsample_param_frame)

## sampling factor

downsample_band_layout[1,1] <- "Downsampling Factor"
downsample_band_layout[2,1] <- gspinbutton(from = 0, to = 1, by = 0.001)
addSpring(downsample_param_frame)

## output params

downsample_output_frame <- gframe(text = "Output Options",
    horizontal = FALSE,
    container = downsample_param_frame,
    expand = FALSE,
    width = 320)

# apply downsample button
downsample_apply_btn <-
gbuiltin("Apply Downsample to Data",
    container = downsample_output_frame,
    handler = function(h,...){
        ds.filename <- svalue(filter_var_filesel)

        ## collect args
        this.args <- list(
            input.table = bquote( # partial dereference
                rbci.env$importlist[[.(ds.filename)]],
                ds.factor = svalue(downsample_band_layout[2,1]),
                time.col = svalue(downsample_varlist)
        )

        ## apply the downsample, add new data file to list
        new.table <-
            list(do.call(downsample.dt,this.args))

        names(new.table) <- paste(ds.filename,
            "downsample", seq_along(new.table),
            sep = ",")
        rbci.env$importlist <- append(rbci.env$importlist,
            new.table)

        ## ensure names are straight
        names(rbci.env$importlist) <-
            make.unique(names(rbci.env$importlist))

        ## add op to reporter
        add.step("downsample.dt", this.args)
    })

    # refresh dataset frame on run
    addHandlerClicked(downsample_apply_btn,
        handler = function(h,...){
            new.datasets <- names(rbci.env$importlist)
            filter_var_filesel[] <- new.datasets
        })

    # alert complete (progress bar?)

    # set some widths (doesn't work if earlier)
    svalue(downsample_param_frame) <- 0.2
}

8.2.28 filter_interface_simple.R

# button pane
simple_pane <- gpanedgroup(horizontal = TRUE,
                           expand = TRUE,
                           fill = TRUE,
                           container = filter_simple_tab)

filter_varlist_frame <- gframe(text = "Apply Columns",
                                horizontal = FALSE,
                                container = simple_pane,
                                expand = TRUE,
                                width = 300)

# populate varlist
filter_varlist <- gcheckboxgroup(
  names(rbci.env$importlist[[svalue(filter_var_filesel, index=TRUE)]]),
  container = filter_varlist_frame,
  use.table = TRUE,
  expand = TRUE)

addHandlerChanged(filter_var_filesel,
  handler = function(h,...) {
    new.dataset.names <-
    names(rbci.env$importlist[[svalue(filter_var_filesel,
                                  index=TRUE)]])
    filter_varlist[] <- new.dataset.names
    filter_grouping_layout[2,1][] <- new.dataset.names
    filter_grouping_layout[4,1][] <- new.dataset.names
  })

simple_filter_pane <- gpanedgroup(horizontal = TRUE,
                                   expand = TRUE,
                                   fill = TRUE,
                                   container = simple_pane)

## filter params
filter_param_frame <- gframe(text = "Filter Parameters",
                              horizontal = FALSE,
                              container = simple_filter_pane,
                              expand = TRUE,
                              width = 300)

# type
filter_type_list <- c("Lowpass","Bandpass","Highpass","Stopband")
filter_type_label <- glabel(text = "Filter Type",
                             container = filter_param_frame)
filter_type_menu <-
gdroplist(filter_type_list,
text = "Filter Type",
container = filter_param_frame,
handler = function (h,...) {
    # enable or disable band param GUI opts on type change
}

# band entry (spinboxes)
filter_band_label <- glabel(text = "Filter Band (Hz)",
    container = filter_param_frame)
filter_band_layout <- glayout(container = filter_param_frame)

# stop band start
filter_band_layout[1,1] <- "Start"
filter_band_layout[2,1] <- gspinbutton(from = 0, to = 4096, by = 0.01)

# stop band end
filter_band_layout[1,2] <- "End"
filter_band_layout[2,2] <- gspinbutton(from = 0, to = 4096, by = 0.01)

filter_band_layout[3,1] <- "Sampling Rate (Hz)"
filter_band_layout[3,2] <- gspinbutton(from = 0.01, to = 4096, by = 0.01)

## application params
filter_grouping_frame <- gframe(text = "Filter Apply Rules",
    horizontal = FALSE,
    container = filter_param_frame,
    expand = TRUE,
    width = 300)

# trial/group vars
filter_grouping_label <- glabel(text = "Data Grouping",
    container = filter_grouping_frame)
filter_grouping_layout <- glayout(container = filter_grouping_frame)

filter_grouping_layout[1,1] <- "First Group (Trial)"
filter_grouping_layout[2,1] <-
gcombobox(
    names(rbci.env$importlist[[svalue(filter_var_filesel, index=TRUE)]])
)

filter_grouping_layout[3,1] <- "Second Group (Channel)"
filter_grouping_layout[4,1] <-
gcombobox(
    names(rbci.env$importlist[[svalue(filter_var_filesel, index=TRUE)]])
)

## output params
filter_output_frame <- gframe(text = "Filter Output Options",
    horizontal = FALSE,
container = filter_param_frame, expand = TRUE, width = 300)
addSpring(filter_output_frame)
# apply filter button
filter_apply_btn <- gbutton("Apply Filter to Data",
  container = filter_output_frame,
  handler = function(h,...){
    ### TODO error checking on filt vals

    filt.type <- svalue(filter_type_menu)
    filt.band <- c(svalue(filter_band_layout[2,1]),
                   svalue(filter_band_layout[2,2]))
    filt.groups <- c(svalue(filter_grouping_layout[2,1]),
                     svalue(filter_grouping_layout[4,1]))
    file.name <- svalue(filter_var_filesel)
    filt.file <- # partial deref
      bquote(rbci.env$importlist[[.((file.name)]])
    filt.srate <- svalue(filter_band_layout[3,2])

    filter.args <- list(
      filter.type = filt.type,
      active.band = filt.band,
      sample.rate = filt.srate,
      filter.groups = filt.groups)

    ## get the designed filter
    my.filter <- do.call(simple.filter, filter.args)
    ## add op to reporter
    add.step("simple.filter", filter.args)

    ## plot the filter as confirmation that it worked
    visible(filter_plot_frame,TRUE)
    filter.plot.args <- list(
      filter = my.filter,
      sample.rate = filt.srate,
      active.band = filt.band)

    print(do.call(plot.filter, filter.plot.args))
    ## add filter to reporter manifest
    add.step("plot.filter", filter.plot.args)

    ## apply the filter, add new data file to list
    ## for reproducibility, add the filter to the list also

    apply.filter.args <- list(
      signal.table = filt.file,
filt.groups = filt.groups,
val.col = svalue(filter_varlist),
filter.obj = my.filter
)

new.table <- list(my.filter,
  do.call(apply.filter, apply.filter.args))
add.step("apply.filter", apply.filter.args)

names(new.table) <- c(paste(file.name, # filter name
  "filter",
  sep = "."),
paste(file.name, # filtered data
  "filtered",
  sep = "."))

rbci.env$importlist <-
  append(rbci.env$importlist, new.table)

## ensure names are straight
names(rbci.env$importlist) <-
  make.unique(names(rbci.env$importlist))
}

## refresh dataset frame on run
addHandlerClicked(filter_apply_btn,
  handler = function(h,...){
    new.datasets <-
      names(rbci.env$importlist)
    filter_var_filesel[] <- new.datasets
  })

## alert complete (progress bar?)

## plot pane
# filter plot on right side
filter_plot_frame <- ggraphics(container = simple_filter_pane)

# set some widths (doesn’t work if earlier)
svalue(filter_param_frame) <- 0.2

8.2.29 import_interface.R

source("./backend/previewers.R")
```r
source("./backend/importers.R")

window.width <- 1000
window.height <- 600
preview.rowlen <- 20

import_win <- gwindow("Data Import",
width = window.width,
height = window.height)

import_tabs <- gnotebook(tab.pos = 2, 
container = import_win)

source("./gwidgets/import_interface_matlab.R")
source("./gwidgets/import_interface_matlab_type2.R")
source("./gwidgets/import_interface_csv.R")
source("./gwidgets/import_interface_rdata.R")
# source("./gwidgets/import_interface_bci2000.R")

8.2.30 import_interface_bci2000.R

bci_tab <- ggroup(container = import_tabs, 
horizontal = FALSE, 
label = "BCI2000")

bci_pane <- gpanedgroup(horizontal = TRUE, 
expand = TRUE, 
fill = TRUE, 
container = bci_tab)

bci_file_frame <- gframe(text = "File path", 
horizontal = FALSE, 
container = bci_pane)
#width = window.width/2)

bci_file_button <- gfilebrowse(text = "", 
container = bci_file_frame, 
quote = FALSE)
addHandlerChanged(bci_file_button, 
handler=function(h,...) { 
  rbci.env$previewfile <- svalue(bci_file_button) 
})

bci_preview_button <- 
gbutton(text = "Preview",
```
container = bci_file_frame,
handler = function(h, ...) {
    rbci.env$importfile <- read.table(rbci.env$previewfile,
        maxLen = 100000)
    # TODO add bci formatting options to GUI
    # delete previous preview frame if present
    if (!"bci_preview_frame" %in% ls()) {
        delete(bci_pane, bci_preview_frame)
    }
    # react to preview type setting
    switch(svalue(bci_preview_type),
        Columnar = {
            # init tabular preview frame
            bci_preview_frame <<- gtable(items =
                as.data.frame(
                    rbci.env$importfile)[seq_len(preview.rowlen),],
                container = bci_pane)
            # column selectors
            # TODO clear previous columns
            rbci.env$columnboxes <<- c()
            for (this.col in
                colnames(as.data.frame(rbci.env$importfile))) {
                rbci.env$columnboxes <<- c(rbci.env$columnboxes,
                    gcheckbox(this.col,
                        checked = FALSE,
                        expand = FALSE,
                        container = bci_option_group))
            }
        },
        Structural = {
            bci_preview_frame <<- gtext(text = paste(capture.output(str( 
                rbci.env$importfile)),"\n"),
                container = bci_pane,
                font.attr=c(family="monospace"))
        },
        Raw = {
            bci_preview_frame <<- gtext(text = paste(capture.output(
                print(sapply(rbci.env$importfile, head,n=2)),"\n"),
                container = bci_pane,
                font.attr=c(family="monospace"))
        }
    }}
bci_preview_optframe <- gframe(text = "Preview Type",
    container = bci_file_frame,
    horizontal = TRUE,
    expand = FALSE)
bci_preview_type <- gdroplist(items = c("Columnar","Structural","Raw"),
    container = bci_preview_optframe,
    fill = "x",
    expand = TRUE)
bci_preview_rownum <- gspinbutton(from = 0, to = 20, by = 1,
    container = bci_preview_optframe)

bci_option_frame <- gframe(text = "Import Columns",
    container = bci_file_frame,
    expand = TRUE)
bci_option_group <- ggroup(use.scrollwindow = TRUE,
    container=bci_option_frame)

bci_export_frame <- gframe(text = "Export/Load",
    container = bci_file_frame)
bci_export_button <- gbutton(text = "Export to .RData",
    type = "save",
    container = bci_export_frame,
    handler = function(h,...) {
        # save file
        # get enabled columns
        colsel <- sapply(rbci.env$columnboxes,svalue)

        # read full-length file
        rbci.env$importfile <-
            readMat(rbci.env$previewfile)

        eegdata <- as.data.table(as.data.frame(
            rbci.env$importfile)[,which(colsel==TRUE)])

        save(eegdata,
            file = gfile(
                filter = list("RData"= list(patterns =
                    c("*.RData")),
                    type = "save")))
    })

bci_load_button <-
gbutton(text = "Import into interface",
        container = bci_export_frame,
        handler = function(h,...) {

    # rename, insert into interface

    # get enabled columns
colsel <- sapply(rbcie.env$columnboxes,svalue)

    # read full-length file
rbcie.env$importfile <-
    readMat(rbcie.env$previewfile)

    # add imported data to list
rbcie.env$importlist[[
    basename(file_path_sans_ext(svalue(bci_file_button)))]]
<- as.data.table(as.data.frame(   rbcie.env$importfile)[,which(colsel==TRUE)])

    # in case of duplicates, mark explicitly
names(rbcie.env$importlist) <-
    make.unique(names(rbcie.env$importlist))

    # TODO cleanup importfile
galert("Import succeeded.",
        title = "Status",
        delay = 2)
})

8.2.31 import_interface_csv.R

csv_tab <- ggroup(container = import_tabs,
        horizontal = FALSE,
        label = "CSV/Text")

csv_pane <- gpanedgroup(horizontal = TRUE,
        expand = TRUE,
        fill = TRUE,
        container = csv_tab)

csv_file_frame <- gframe(text = "File path",
        horizontal = FALSE,
        container = csv_pane)
        #width = window.width/2)

csv_file_button <- gfilebrowse(text = "",
        container = csv_file_frame,
        ...)
quote = FALSE

addHandlerChanged(csv_file_button,
    handler=function(h,...) {
        rbci.env$previewfile <- svalue(csv_file_button)
    })

csv_preview_button <- gbutton(text = "Preview",
    container = csv_file_frame,
    handler = function(h, ...) {
        rbci.env$importfile <- read.table(rbci.env$previewfile,
            n rows = preview.rowlen)

        ## TODO add CSV formatting options to GUI
        ## delete previous preview frame if present
        if ("csv_preview_frame" %in% ls()) {
            delete(csv_pane, csv_preview_frame)
        }

        ## react to preview type setting
        switch(svalue(csv_preview_type),
            Columnar = {
                ## init tabular preview frame
                csv_preview_frame <<- gtable(items =
                    as.data.frame(
                        rbci.env$importfile)[seq_len(preview.rowlen),],
                    container = csv_pane)

                ## column selectors
                if(exists("csv_column_sel")) {
                    csv_column_sel[] <<- # replace members if exists already
                        colnames(as.data.frame(rbci.env$importfile))
                } else {
                    csv_column_sel <-
                        gcheckboxgroup( # create if new
                        colnames(as.data.frame(rbci.env$importfile)),
                        container = csv_option_group,
                        use.table = TRUE,
                        expand = TRUE)
                }
            },
            Structural = {
                csv_preview_frame <<-
                    gtext(text = paste(capture.output(str(
                        rbci.env$importfile)),"\n"),
                    container = csv_pane,
                    font.attr=c(family="monospace"))
            },
            Raw = {
        )

145
```r
csv_preview_frame <-
  gtext(text = paste(capture.output(
    print(sapply(rbci.env$importfile, head, n=2)), "\n"),
    container = csv-pane,
    font.attr=c(family="monospace"))
  }
}

csv_preview_optframe <- gframe(text = "Preview Type",
  container = csv_file_frame,
  horizontal = TRUE,
  expand = FALSE)

csv_preview_type <- gdroplist(items = c("Columnar","Structural","Raw"),
  container = csv_preview_optframe,
  fill = "x",
  expand = TRUE)

csv_preview_rownum <- gspinbutton(from = 0, to = 20, by = 1,
  container = csv_preview_optframe)

csv_option_frame <- gframe(text = "Import Columns",
  container = csv_file_frame,
  horizontal = FALSE,
  expand = TRUE)

csv_option_group <- ggroup(use.scrollwindow = TRUE,
  container = TRUE,
  expand = TRUE,
  container=csv_option_frame)

csv_export_frame <- gframe(text = "Export/Load",
  container = csv_file_frame)

csv_export_button <- gbutton(text = "Export to R Data",
  type = "save",
  container = csv_export_frame,
  handler = function(h,...) {
    # save file
    # get enabled columns
colsel <- sapply(rbci.env$columnboxes,svalue)

    # read full-length file
    rbci.env$importfile <-
      read.table(rbci.env$previewfile)

eegdata <- as.data.table(as.data.frame(
    rbci.env$importfile)[,which(colsel==TRUE)])
```
```r
save(eegdata,
    file = gfile(
        filter = list("R Data (RData)" = list(patterns = c("*.RData")),
                    type = "save"))
}

csv_load_button <-
gbutton(text = "Import into interface",
    container = csv_export_frame,
    handler = function(h, ...) {
        # rename, insert into interface

        # get enabled columns
colsel <- sapply(rbci.env$columnboxes,svalue)

        # read full-length file
        rbci.env$importfile <-
            read.table(rbci.env$previewfile)

        # add imported data to list
        rbci.env$importlist[[
            basename(file_path_sans_ext(svalue(csv_file_button))))] <-
            as.data.table(as.data.frame(
                rbci.env$importfile)[,which(colsel==TRUE)])

        # in case of duplicates, mark explicitly
        names(rbci.env$importlist) <-
            make.unique(names(rbci.env$importlist))

        # TODO cleanup importfile

galert("Import succeeded.",
        title = "Status",
        delay = 2)
})
```

---

### 8.2.32 import_interface_matlab.R

```r
matlab_tab <- ggroup(container = import_tabs,
    horizontal = FALSE,
    label = "MATLAB")

matlab_pane <- gpanedgroup(horizontal = TRUE,
    expand = TRUE,
    fill = TRUE,
    container = matlab_tab)
```
```r
matlab_file_frame <- gframe(text = "File path",
    horizontal = FALSE,
    container = matlab_pane)

# width = window.width/2)

matlab_file_button <- gfilebrowse(text = "",
    container = matlab_file_frame,
    quote = FALSE)

addHandlerChanged(matlab_file_button,
    handler = function(h, ...)
    { 
        rbci.env$previewfile <- svalue(matlab_file_button)
    })

matlab_preview_button <-
    gbutton(text = "Preview",
        container = matlab_file_frame,
        handler = function(h, ...) 
        { 
            rbci.env$importfile <- readMat(rbci.env$previewfile,
                maxLength = 100000)

            ## delete previous preview frame if present
            if ("matlab_preview_frame" %in% ls()) {
                delete(matlab_pane, matlab_preview_frame)
            }

            ## react to preview type setting
            switch(svalue(matlab_preview_type),
                Columnar = {
                    ## init tabular preview frame
                    matlab_preview_frame <-
                        gtable(items =
                            as.data.frame(
                                rbci.env$importfile)[seq_len(preview.rowlen),],
                            container = matlab_pane)

                    ## column selectors
                    if (exists("matlab_column_sel")) {
                        matlab_column_sel[] <- # replace members if exists
                            colnames(as.data.frame(rbci.env$importfile))
                    } else {
                        matlab_column_sel[<- # global assign to resolve scope
                            gcheckboxgroup( # create if new
                                colnames(as.data.frame(rbci.env$importfile)),
                                container = matlab_option_group,
                                use.table = TRUE,
                                expand = TRUE)
                    }
                },
                Structural = {
```
matlab_preview_frame <-
  gtext(text = paste(capture.output(str(
    rbci.env$importfile)),"
"),
    container = matlabPane,
    font.attr = c(family = "monospace"))
},
Raw = {
  matlab_preview_frame <-
  gtext(text = paste(capture.output(
    print(sapply(rbci.env$importfile,head,n=2))),"
"),
    container = matlabPane,
    font.attr = c(family = "monospace"))
}
)
)

matlab_preview_optframe <- gframe(text = "Preview Type",
  container = matlabFileFrame,
  horizontal = TRUE,
  expand = FALSE)
matlab_preview_type <- gdroplist(items = c("Columnar","Structural","Raw"),
  container = matlab_preview_optframe,
  fill = "x",
  expand = TRUE)
matlab_preview_rownum <- gspinbutton(from = 0, to = 20, by = 1,
  container = matlab_preview_optframe)

matlab_option_frame <- gframe(text = "Import Columns",
  container = matlabFileFrame,
  horizontal = FALSE,
  expand = TRUE)
matlab_option_group <- ggroup(use.scrollwindow = TRUE,
  container = matlab_option_frame)

matlab_export_frame <- gframe(text = "Export/Load",
  container = matlabFileFrame)

matlab_export_button <-
gbutton(text = "Export to .RData",
  type = "save",
  container = matlab_export_frame,
  handler = function(h,...) {
    # save file
    # get enabled columns
colsel <- sapply(rbci.env$columnboxes,svalue)
# read full-length file
rbci.env$importfile <-
  readMat(rbci.env$previewfile)

eegdata <- as.data.table(as.data.frame(
  rbci.env$importfile)[, which(colsel == TRUE)])

save(eegdata,
  file = gfile(
    filter = list("RData" = list(patterns = c("*.RData")),
    type = "save"))
)

matlab_load_button <-
gbutton(text = "Import into interface",
  container = matlab_export_frame,
  handler = function(h,...) {
    # rename, insert into interface
    # get enabled columns
    colsel <- sapply(rbci.env$columnboxes,svalue)
    # read full-length file
    rbci.env$importfile <-
      readMat(rbci.env$previewfile)
    # add imported data to list
    rbci.env$importlist[[
      basename(file_path_sans_ext(svalue(matlab_file_button)))]]
      <- as.data.frame(
        rbci.env$importfile)[, which(colsel == TRUE)])
    # in case of duplicates, mark explicitly
    names(rbci.env$importlist) <-
      make.unique(names(rbci.env$importlist))
    galert("Import succeeded.",
      title = "Status",
      delay = 2)
  })

---

### 8.2.33 import_interface_matlab_type2.R

matlab_type2_tab <- ggroup(container = import_tabs,
  horizontal = FALSE,
  label = "MATLAB (Type 2)"
)
matlab_type2_pane <- gpanedgroup(horizontal = TRUE,
    expand = TRUE,
    fill = TRUE,
    container = matlab_type2_tab)

matlab_type2_file_frame <- gframe(text = "File path",
    horizontal = FALSE,
    container = matlab_type2_pane)

matlab_type2_file_button <- gfilebrowse(text = "",
    container = matlab_type2_file_frame,
    quote = FALSE)
addHandlerChanged(matlab_type2_file_button,
    handler=function(h,...) {
        rbci.env$previewfile <- svalue(matlab_type2_file_button) 
    })

matlab_type2_preview_button <-
    gbutton(text = "Preview",
        container = matlab_type2_file_frame,
        handler = function(h, ...) {
            rbci.env$importfile <- readMat(rbci.env$previewfile,
                maxLength = 100000)
            eegdata <-
                matlab_type2_import(rbci.env$importfile,
                    eeg.ind = svalue(matlab_type2_eegindex),
                    tgt.ind = svalue(matlab_type2_tgtindex))
            # delete previous preview frame if present
            if ("matlab_type2_preview_frame" %in% ls()) {
                delete(matlab_type2_pane, matlab_type2_preview_frame)
            }
            # react to preview type setting
            switch(svalue(matlab_type2_preview_type),
                Structural = {
                    matlab_type2_preview_frame <-
                        gtext(text = paste(capture.output(str(
                        rbci.env$importfile)),"\n"),
                        container = matlab_type2_pane,
                        font.attr=c(family="monospace"))
                },
                Graphical = {
                    matlab_type2_preview_frame <-
                        ggraphics(container = matlab_type2_pane)
                }
            # visible(matlab_type2_preview_frame) <- TRUE
            # print(eegdata$preview.plot) # should plot here
\begin{verbatim}
print(grand.means.plot(eegdata, plot.title = basename(svalue(matlab_type2_file_button)))
}
}

matlab_type2_preview_optframe <- gframe(text = "", container = matlab_type2_file_frame, horizontal = FALSE, expand = FALSE)
glabel("Preview Type", container = matlab_type2_preview_optframe)
matlab_type2_preview_type <- gdroplist(items = c("Structural", "Graphical"), container = matlab_type2_preview_optframe, fill = "x", expand = TRUE)
glabel("EEG index", container = matlab_type2_preview_optframe)
matlab_type2_eegindex <- gspinbutton(from = 1, container = matlab_type2_preview_optframe)
glabel("Class label index", container = matlab_type2_preview_optframe)
matlab_type2_tgtindex <- gspinbutton(from = 1, container = matlab_type2_preview_optframe)

addSpring(matlab_type2_file_frame)

matlab_type2_export_optframe <- gframe(text = "Export/Load", horizontal = TRUE, expand = FALSE, container = matlab_type2_file_frame)

matlab_type2_export_button <- gbutton(text = "Export to .RData", type = "save", container = matlab_type2_export_optframe, handler = function(h,...) {
    # read full-length file
    rbci.env$importfile <- readMat(rbci.env$previewfile)
eegdata <-
    matlab_type2_import(rbci.env$importfile, eeg.ind = svalue(matlab_type2_eegindex), tgt.ind = svalue(matlab_type2_tgtindex))
    save(eegdata,}
\end{verbatim}
file = gfile(
    filter = list("RData"= list(patterns = c("*.RData")),
        type = "save"))
)

matlab_type2_load_button <-
gbutton(text = "Import into interface",
    container = matlab_type2_export_frame,
    handler = function(h,...) {
        # rename, insert into interface
        # get enabled columns
        # colsel <- sapply(rbci.env$columnboxes,svalue)
        # read full-length file
        rbci.env$importfile <-
            readMat(rbci.env$previewfile)

        # add imported data to list
        rbci.env$importlist[[
            basename(file_path_sans_ext(svalue(matlab_type2_file_button)))]]
            <-
            matlab_type2_import(rbci.env$importfile,
                eeg.ind = svalue(matlab_type2_eegindex),
                tgt.ind = svalue(matlab_type2_tgtindex))
        # in case of duplicates, mark explicitly
        names(rbci.env$importlist) <-
            make.unique(names(rbci.env$importlist))
        galert("Import succeeded.",
            title = "Status",
            delay = 2)
    })

8.2.34 import_interface_rdata.R

```r
rdata_tab <- ggroup(container = import_tabs,
        horizontal = FALSE,
        label = "R Data")

rdata_pane <- gpanedgroup(horizontal = TRUE,
        expand = TRUE,
        fill = TRUE,
        container = rdata_tab)
```
rdata_file_frame <- gframe(text = "File path",
    horizontal = FALSE,
    container = rdata_pane)
#width = window.width/2)

rdata_file_button <- gfilebrowse(text = "",
    container = rdata_file_frame,
    quote = FALSE)

addHandlerChanged(rdata_file_button,
    handler = function(h,...) {
        rbci.env$previewfile <- svalue(rdata_file_button)
    })

rdata_preview_button <-
gbutton(text = "Preview",
    container = rdata_file_frame,
    handler = function(h,...) {
        rbci.env$importfile <- load_obj(rbci.env$previewfile)
        # nrows = preview.rowlen)
        # delete previous preview frame if present
        if ("rdata_preview_frame" %in% ls()) {
            delete(rdata_pane, rdata_preview_frame)
        }
        ## react to preview type setting
        switch(svalue(rdata_preview_type),
            Columnar = {
                # init tabular preview frame
                rdata_preview_frame <<- gtable(items =
                    as.data.frame(rbcie.env$importfile)[seq_len(preview.rowlen),],
                    container = rdata_pane)
                ## column selectors
                if (exists("rdata_column_sel")) {
                    rdata_column_sel[] <- # replace members if exists
                    colnames(as.data.frame(rbcie.env$importfile))
                } else {
                    rdata_column_sel <<- # global assign to resolve scope
                    gcheckboxgroup( # create if new
                        colnames(as.data.frame(rbcie.env$importfile)),
                        container = rdata_option_group,
                        use.table = TRUE,
                        expand = TRUE)
                }
            },
            Structural = {
        },
rdata_preview_frame <-
gtext(text = paste(capture.output(str(
  rbci.env$importfile)),"\n"),
  container = rdata-pane,
  font.attr=c(family="monospace"))
},
Raw = {
  rdata_preview_frame <-
gtext(text = paste(capture.output(
    print(sapply(rbci.env$importfile,head,n=2))),"\n"),
  container = rdata-pane,
  font.attr=c(family="monospace"))
}
}

rdata_preview_optframe <- gframe(text = "Preview Type",
  container = rdata_file_frame,
  horizontal = TRUE,
  expand = FALSE)
rdata_preview_type <- gdroplist(items = c("Columnar","Structural","Raw"),
  container = rdata_preview_optframe,
  fill = "x",
  expand = TRUE)
rdata_preview_rownum <- gspinbutton(from = 0, to = 20, by = 1,
  container = rdata_preview_optframe)

rdata_option_frame <- gframe(text = "Import Columns",
  horizontal = FALSE,
  container = rdata_file_frame,
  expand = TRUE)
rdata_option_group <- ggroup(use.scrollwindow = TRUE,
  container=rdata_option_frame)

rdata_export_frame <- gframe(text = "Export/Load",
  horizontal = TRUE,
  expand = FALSE,
  container = rdata_file_frame)

rdata_export_button <- gbutton(text = "Export to R Data (RData)",
  type = "save",
  container = rdata_export_frame,
  handler = function(h,...) {
    # save file
    # get enabled columns
colsel <- sapply(rbci.env$columnboxes,svalue)
# read full-length file
rbci.env$importfile <-
  load_obj(rbci.env$previewfile)

eegdata <-
  rbci.env$importfile

save(eegdata,
  file = gfile(
    filter = list("RData"=
      list(patterns =
        c("*.RData")),
      type = "save"))
)

rdata_load_button <-
gbutton(text = "Import into interface",
  container = rdata_export_frame,
  handler = function(h,...) {

    # get enabled columns
    colsel <- sapply(rbci.env$columnboxes,svalue)

    # read full-length file
    rbci.env$importfile <-
      load_obj(rbci.env$previewfile)

    ## add imported data to list
    rbci.env$importlist[[
      basename(file_path_sans_ext(svalue(rdata_file_button))))]
     <-
      rbci.env$importfile[,which(colsel==TRUE), with = FALSE]
    # in case of duplicates, mark explicitly
    names(rbci.env$importlist) <-
      make.unique(names(rbci.env$importlist))

    # TODO cleanup importfile

galert("Import succeeded.",
  title = "Status",
  delay = 2)
})

8.2.35 init_interface.R

source("./backend/init_backend.R")
options("guiToolkit"="RGtk2")
init_win <- gwindow("R BCI",
               width = 200)

init_btn_group <- ggroup(container = init_win,
                         horizontal = FALSE)

import_btn <- gbutton(
    text = "Import/Load Data",
    container = init_btn_group,
    handler = function(h, ...)
    {
        source("./gwidgets/import_interface.R")
    })

explain_btn <- gbutton(
    text = "Explore Data",
    container = init_btn_group,
    handler = function(h, ...)
    {
        source("./gwidgets/explore_interface.R")
    })

filter_btn <- gbutton(
    text = "Filter Data",
    container = init_btn_group,
    handler = function(h, ...)
    {
        source("./gwidgets/filter_interface.R")
    })

transform_btn <- gbutton(
    text = "Transforms/Unsupervised Learning",
    container = init_btn_group,
    handler = function(h, ...)
    {
        source("./gwidgets/trans_interface.R")
    })

classify_btn <- gbutton(
    text = "Classification/Supervised Learning",
    container = init_btn_group,
    handler = function(h, ...)
    {
        source("./gwidgets/class_interface.R")
    })

report_btn <- gbutton(}
text = "Review/Generate Reports",
container = init_btn_group,
handler = function(h, ...) {
  source("./gwidgets/report_interface.R")
}

opts_btm <- gbutton(
  text = "Parallelization Options",
  container = init_btn_group,
  handler = function(h, ...)
  {
    source("./gwidgets/opts_interface.R")
  }
)

tool_btm <- gbutton(
  text = "Custom Tool Designer",
  container = init_btn_group,
  handler = function(h, ...)
  {
    source("./gwidgets/tool_interface.R")
  }
)

quit_btm <- gbutton(
  text = "Quit",
  container = init_btn_group,
  handler = function(h, ...)
  {
    dispose(init_win)
  })

8.2.36  opts_interface.R

window.width <- 450
window.height <- 500

opts_win <- gwindow("Parallelization Options",
  width = window.width,
  height = window.height)

parallel_frame <- gframe(container = opts_win,
  horizontal = FALSE)

parallel_label <- glabel("Cluster Type",
  container = parallel_frame)
parallelbackend_type_list <- c("multicore (local)", "SNOW", "Redis")
parallelbackend_type_menu <-
gdroplist(parallelbackend_type_list,
text = "Backend Type",
container = parallel_frame,
handler = function (h,...) {
    # enable or disable param GUI opts on type change

    ## param names by type
    # sigma for rbf/Laplace
    # degree, scale, offset for Polynomial
    # scale, offset for tanhdot
    # sigma, order, degree, for Bessel
    # sigma, degree for ANOVA
    switch (svalue(h$obj),
        "multicore (local)" = {
            ## load multicore subGUI
            source('./gwidgets/opts_interface_multicore.R')
        },
        "SNOW" = {
            ## load snow subGUI
            source('./gwidgets/opts_interface_snow.R')
        },
        "Redis" = {
            ## load redis subGUI
            source('./gwidgets/opts_interface_redis.R')
        }
    )
}
## since multicore is default and droplist handler won’t run then, we manually
## call default subGUI
source('./gwidgets/opts_interface_multicore.R')

8.2.37 opts_interface_multicore.R

multicore_backendopts_frame <- gframe(horizontal = FALSE,
text = "Multicore options")

## initialize lower layout again
if (exists('backendopts_frame') && isExtant(backendopts_frame)) {
    delete(parallel_frame,
        backendopts_frame)
}

add(parallel_frame, multicore_backendopts_frame)

backendopts_frame <- multicore_backendopts_frame # add simple name reference
numcores_label <- glabel("# of Cores",
    container = backendopts_frame)

if (detectCores() < 2) { # single core case
    numcores_nope <- glabel(
        "Single core machine detected: multicore unavailable",
        container = backendopts_frame)
} else {
    numcores_slider <- gslider(from = 1, to = detectCores(), by = 1,
        handler = function(h,...){
            # set number of cores
            rbci.env$numcores <- svalue(h$obj)
        })
}

8.2.38 opts_interface_redis.R

build_redisfields <- function(num.clusters, oldframe, container) {
    ### Builds form layout for Redis config
    if (exists('redisopts_layout') & isExtant(redisopts_layout)) {
        delete(oldframe, redisopts_layout)
    }
    redisopts_layout <- glayout()

    refresh.widget(container, oldframe) # delete old layout

    sapply(seq_len(num.clusters), function(this.row) {
        redisopts_layout[this.row,1] <- paste("Queue",this.row)
        redisopts_layout[this.row,2] <- glabel("Hostname",
            editable = TRUE)
        redisopts_layout[this.row,3] <- glabel("Port",
            editable = TRUE)
        redisopts_layout[this.row,4] <- glabel("Password",
            editable = TRUE)

        return()
    })
    add(oldframe, redisopts_layout)
}

redis_backendopts_frame <- gframe(horizontal = FALSE,
    text = "Redis queue options")
## initialize lower layout again
refresh.widget(parallel_frame, backendopts_frame, redis_backendopts_frame)
backendopts_frame <- redis_backendopts_frame # add simple name reference
redis_numclust <- gspinbutton(from = 1, to = 16, value = 1, 
                          container = backendopts_frame)

redis_cfg_btn <-
gbutton("Configure queues", 
        container = backendopts_frame, 
        handler = function(h,...) {

    build_redisfields(svalue(redis_numclust), 
                     backendopts_frame, 
                     parallel_frame)

})

redis_control_frame <- gframe("Queue control", 
                             container = backendopts_frame, 
                             center = TRUE)

redis_start_btn <-
gbutton("Register Queues", 
        container = redis_control_frame, 
        handler = function(h,...) {

    queue.optslist <- vector("list", # hack: we know how many columns in layout 
                              length(redisopts_layout[])/4)

    for (this.row in seq_along(queue.optslist)) {
        queue.optslist[[this.row]] <-
        list(queue = svalue(redisopts_layout[this.row,1]), 
             hostname = svalue(redisopts_layout[this.row,2]), 
             port = svalue(redisopts_layout[this.row,3]), 
             password = svalue(redisopts_layout[this.row,4]))

    }

    ### TODO error checking on opts (fails silently if malformed)
    rbci.env["redisqueue"] <- # return queue names for later
                               # deregistration
    lapply(queue.optslist, function(this.queue) {
        registerDoRedis(unlist(this.queue))
        return(this.queue["queue"])
    })

})

redis_stop_btn <-
gbutton("Stop Queues", 
        container = redis_control_frame, 
        handler = function(h,...) {

    lapply(rbci.env["redisqueue"], removeQueue)
    ### TODO add error handling for queue status

})

8.2.39  opts_interface_snow.R
build_snowfields <- function(num.clusters, oldframe, container) {
  ## Builds form layout for SNOW cluster config
  if (exists("snowopts_layout") && isExtant(snowopts_layout)) {
    delete(oldframe, snowopts_layout)
  }
  snowopts_layout <<- glayout()

  refresh.widget(container, oldframe) # delete old layout

  sapply(seq_len(num.clusters), function(this.row) {
    snowopts_layout[this.row,1] <- paste("Machine",this.row)
    snowopts_layout[this.row,2] <- glabel("Hostname",
        editable = TRUE)
    snowopts_layout[this.row,3] <- glabel("RScript path",
        editable = TRUE)
    snowopts_layout[this.row,4] <- glabel("SNOW library path",
        editable = TRUE)
    return()
  })
  add(oldframe, snowopts_layout)
}

snow_backendopts_frame <- gframe(horizontal = FALSE,
  text = "SNOW cluster options")
## initialize lower layout again
refresh.widget(parallel_frame, backendopts_frame, snow_backendopts_frame)
backendopts_frame <- snow_backendopts_frame # add simple name reference

## static options
staticopts_frame <- gframe(container = backendopts_frame,
    horizontal = TRUE)

snow_numlabel <- glabel("Number of workstations",
    container = staticopts_frame)
snow_numclust <- gspinbutton(from = 1, to = 16, value = 1,
    container = staticopts_frame)

snow_typelabel <- glabel("Interface type",
    container = staticopts_frame)

snow_typelist <- c("SOCK","MPI")
snow_type <- gdroplist(snow_typelist,
    container = staticopts_frame)

snow_cfg_btn <-
  gbutton("Configure clusters",
    container = backendopts_frame,
47 handler = function(h,...) {
48  ## generate dynamic options
49  build_snowfields(svalue(snow_numclust),
50      backendopts_frame,
51      parallel_frame)
52  }
53 }
54
55 snow_control_frame <- gframe("Cluster control",
56      container = backendopts_frame,
57      center = TRUE)
58
59 snow_start_btn <-
60  gbutton("Start Cluster",
61      container = snow_control_frame,
62      handler = function(h,...) {
63        cluster.optslist <-
64            vector("list", # hack: we know how many columns in layout
65            length(snowopts_layout[])/4)
66            for (this.row in seq_along(cluster.optslist)) {
67              cluster.optslist[[this.row]] <-
68                  list(host = svalue(snowopts_layout[this.row,2]),
69                      rscript = svalue(snowopts_layout[this.row,3]),
70                      snowlib = svalue(snowopts_layout[this.row,4]))
71          }
72          ### TODO error checking on opts (fails silently if malformed)
73          rbci.env$cluster <-
74              makeCluster(cluster.optslist,
75                  type = svalue(snow_type))
76              registerDoSNOW(rbci.env$cluster)
77          }
78 }
79
80 snow_stop_btn <-
81  gbutton("Stop Cluster",
82      container = snow_control_frame,
83      handler = function(h,...){
84        stopCluster(rbci.env$cluster)
85      })
86  

8.2.40 report_interface.R

window.width <- 1000
window.height <- 600
preview.rowlen <- 20
report_win <- gwindow("Report Generator",
                      width = window.width,
reportPane <- gpanedgroup(horizontal = TRUE,
                         expand = TRUE,
                         fill = TRUE,
                         container = reportWin)

processGroup <- ggroup(use.scrollwindow = TRUE,
                        horizontal = FALSE,
                        expand = TRUE,
                        container = reportPane,
                        width = 200)

processFrame <- gframe(text = "Processed Steps",
                        horizontal = FALSE,
                        container = processGroup,
                        expand = TRUE)

## populate step selector
processStepSel <- gtable(tabulate.steplist(rbci.env$steplist),
                          container = processFrame,
                          use.table = TRUE,
                          expand = TRUE)

## update preview text, summary on selection change
addHandlerChanged(processStepSel,
                   handler = function(h,...) {
                     codeText <-
                     processStepSel[svalue(processStepSel,index=TRUE),
                                     'code']

                     summaryText <-
                     processStepSel[svalue(processStepSel,index=TRUE),
                                    'summary']

                     svalue(stepCodeText) <-
                     codeText

                     svalue(stepSummary) <-
                     paste("Summary: ",summaryText)
                   })

## controls for changing step ordering/enabledness
processStepUp <- gbutton(
                      text = "",
                      container = processFrame,
                      handler = function(h,...) {
                        scoot.gtable.row(processStepSel,
                                         svalue(processStepSel, index = TRUE),
                                         "up")
                      })
process_step_down <-
  gbutton(text = "", 
          container = process_frame, 
          handler = function(h, ...) {
            scoot.gtable.row(process_step_sel, 
                             svalue(process_step_sel, index = TRUE), 
                             "down")
          })
process_step_toggle <-
  gbutton(text = "Enable/Disable", 
          container = process_frame, 
          handler = function(h, ...) {
            row.ind <- svalue(process_step_sel, index = TRUE)
            toggle.row(process_step_sel, # sets GUI part
                        row.ind)
            ## sets actual env list part
            rbci.env$steplist[[row.ind]]$enabled <- TRUE
          })

# addSpring(process_frame)

## output options
# directory name
report_output_sel <- gfilebrowse(text = "Output directory name", 
                                   type = "selectdir", 
                                   container = process_frame)

report_output_layout <- glayout(container = process_frame)

report_output_layout[1,1] <-
  gbutton(text = "Generate Report", 
          handler = function(h,...) {

            report.steps <- process_step_sel
            report.title <- svalue(report_opts_title)
            report.auth <- svalue(report_opts_author)
            report.dir <- svalue(report_output_sel)
            report.knit <- svalue(report_output_layout[1,2])

            build.report(report.steps, report.title, 
                         report.auth, report.dir, report.knit)
          })

report_output_layout[1,2] <-
  gcheckbox("Run exported report and remove disabled steps?"

# Load subitems (into tabs)
source("./gwidgets/report_interface_text.R")

# set some widths (doesn't work if earlier)
svalue(report_pane) <- 0.4

8.2.41 report_interface_graphics.R

graphics-pane <- gpanedgroup(horizontal = TRUE,
   expand = TRUE,
   fill = TRUE,
   container = graphics-tab)

graphics-opt-frame <- gframe("Output Options",
   horizontal = FALSE,
   container = graphics-pane)

glabel("Size", container = graphics-opt-frame)
graphics-opt-layout <- glayout(container = graphics-opt-frame)

# picture sizes
graphics-opt-layout[1,1] <- "Width"
graphics-opt-layout[2,1] <- gspinbutton(from = 200, to = 1920,
   by = 1, value = 320)

graphics-opt-layout[1,2] <- "Height"
graphics-opt-layout[2,2] <- gspinbutton(from = 200, to = 1080,
   by = 1, value = 240)

# graphic-opt-verbose <- gdroplist(verbosity-levels,
#   container = graphic-opt-frame)
#
# glabel("Format", container = graphic-opt-frame)
# format-types <- c("Raw", "Markdown", "HTML")
# graphic-opt-format <- gdroplist(format-types,
#   container = graphic-opt-frame)

addSpring(graphics-opt-frame)
graphics-reload-button <- gbutton("Update Preview",
   container = graphics-opt-frame)

graphics-output-preview <- ggraphics(container = graphics-pane,
   expand = TRUE)

8.2.42 report_interface_script.R
8.2.43 report_interface_text.R

```r
# text_pane <- gpanedgroup(horizontal = FALSE,
#                          expand = TRUE,
#                          fill = TRUE,
#                          container = report_pane)

text_pane_group <- ggroup(container = report_pane,
                          horizontal = FALSE)

step_summary <- glabel("Summary: ",
                        container = text_pane_group)

step_code_frame <- gframe("Section Code",
                          horizontal = FALSE,
                          container = text_pane_group,
                          expand = TRUE)

step_code_text <- gtext(container = step_code_frame, # step_code_frame,
                        font.attr = c(family="monospace"),
                        expand = TRUE)

report_opts_frame <- gframe("Report Options",
                          horizontal = TRUE,
                          container = text_pane_group)

report_opts_title <- glabel("Report Title",
                          editable = TRUE,
                          container = report_opts_frame,
                          expand = TRUE)

report_opts_author <- glabel("Report Author",
                          editable = TRUE,
                          container = report_opts_frame,
                          expand = TRUE)
```

8.2.44 tool_interface.R
window.width <- 1000
window.height <- 600

tool_win <- gwindow("Tool Designer",
    width = window.width,
    height = window.height)

tool_pane <- gpanedgroup(horizontal = TRUE,
    expand = TRUE,
    fill = TRUE,
    container = tool_win)

tool_var_group <- ggroup(use.scrollwindow = TRUE,
    horizontal = FALSE,
    expand = TRUE,
    container=tool_pane,
    width = 200)
tool_var_frame <- gframe(text = "Scripts",
    horizontal = FALSE,
    container = tool_var_group,
    expand = TRUE)

tool_edit_frame <- ggroup(text = "Editor",
    container = tool_pane,
    use.scrollwindow = FALSE,
    horizontal = FALSE,
    expand = TRUE)

tool_edit_text <- gtext(text =
    "tool1 <- function(input.table = rbci.env$selected_data, ...) {

    ## EDIT HERE

}"
    ,
    container = tool_edit_frame,
    height = 550, width = 550,
    font.attr = c(family="monospace"))

## script directory selector

## EXECUTE FUNCTION ON DATA

tool1()

## script directory selector
tool_dir_button <- gfilebrowse(text = "Script Directory",
    type = "selectdir",
    container = tool_var_frame,
    quote = FALSE)
## set initial directory to pwd
svalue(tool_dir_button) <- getwd()
## function to update script list
## TODO move to separate file
## Buttons that add new things should refresh the dataset selector

addHandlerChanged(tool_dir_button,
    handler = function(h,...) {
        new.scripts <-
        dir(svalue(tool_dir_button),
            pattern = "*.R")
        tool_var_scriptsel[] <- new.scripts
    })

## initialize script list for placement
script_list_frame <- gframe(text = "Scripts",
    expand = TRUE,
    horizontal = FALSE,
    container = tool_var_frame)
tool_var_scriptsel <- gradio(dir(svalue(tool_dir_button),
    pattern = "*.R"),
    container = script_list_frame)

tool_loadsave_frame <- gframe(text = "Load/Save",
    container = tool_var_frame)
tool_load_button <-
gbutton(text = "Edit Selected Script",
    container = tool_loadsave_frame,
    handler = function (h,...) {
        ## TODO add file directory
        ## load script file
        load.name <- svalue(tool_var_scriptsel)
        load.dir <- svalue(tool_dir_button)
        load.path <- paste(load.dir, load.name, sep="/"
        ## get text
        load.text <-
        readChar(load.path,
            file.info(load.path)$size)
        ## make available to GUI
        svalue(tool_edit_text) <- load.text
    })

tool_save_button <-


gbutton(text = "Save Script",
type = "save",
container = tool_loadsave_frame,
handler = function (h,...) {
    file.text <-
        svalue(tool_edit_text)

    write(file.text,
        file = gfile(
            filter =
                list("R scripts" = list(patterns = (*.R))),
                type = "save"))
})
addHandlerClicked(tool_save_button,
    handler = function(h,...){
new.scripts <-
    dir(svalue(tool_dir_button),
        pattern = ".R")
    tool_var_scriptsel[] <- new.scripts
})
addSpring(tool_edit_frame)
tool_edit_runframe <- gframe(text = "Run/Output",
    horizontal = TRUE,
    container = tool_edit_frame)
tool_run_button <-
gbutton(text = "Run Loaded Script",
    container = tool_edit_runframe,
    anchor = c(1,1),
    handler = function (h,...) {
## get data
    rbci.env$selected_data <-
        rbci.env$importlist[svalue(tool_input_name)]

## do the call, add to dataset list
## save output var to env if successful
    rbci.env$importlist[svalue(tool_output_name)] <-
        eval(parse(text = svalue(tool_edit_text)))

## add string manually to reporter
## TODO refactor this to match other steps
new.step <-
    list(summary = paste("custom function of ",
        svalue(tool_input_name)),
        enabled = FALSE,
        code = svalue(tool_edit_text))
    rbci.env$steplist <- append(rbci.env$steplist,
        list(new.step))
## print error in console if not (automatic?)
### TODO notify w/ alert

## update available data
```r
tool_input_name[] <- names(rbci.env$importlist)
```
```r
tool_input_name <- gdroplist(names(rbci.env$importlist),
                              container = tool_edit_runframe)
```
```r
tool_output_name <- gedit(text = "Output.Variable",
                          container = tool_edit_runframe,
                          width = 25)
```

# set some widths (doesn’t work if earlier)
```r
svalue(toolPane) <- 0.39
```

8.2.45 trans_interface.R

```r
window.width <- 1000
window.height <- 600

trans_win <- gwindow("Transform/Cluster Data",
                     width = window.width,
                     height = window.height)

trans_pane <- gpanedgroup(horizontal = TRUE,
                            expand = TRUE,
                            fill = TRUE,
                            container = trans_win)

trans_var_group <- ggroup(use.scrollwindow = TRUE,
                           horizontal = FALSE,
                           expand = TRUE,
                           container = trans_pane,
                           width = 200)

trans_var_frame <- gframe(text = "Source Data",
                          horizontal = FALSE,
                          container = trans_var_group,
                          expand = TRUE)

## populate data set selector
```r
if (!is.null(names(rbci.env$importlist))) {
    trans_var_filesel <- gradio(names(rbci.env$importlist),
                                container = trans_var_frame)
} else {
```
trans_var_filesel <- glabel("No data found.",
    container = trans_var_frame)
}

trans_task_book <- gnotebook(tab.pos = 3,
    container = trans_pane)

trans_pca_tab <- gframe(label = "PCA",
    container = trans_task_book)

trans_csp_tab <- gframe(label = "CSP",
    container = trans_task_book)

trans_kmeans_tab <- gframe(label = "k-means",
    container = trans_task_book)

source("./gwidgets/trans_interface_pca.R")
source("./gwidgets/trans_interface_csp.R")
source("./gwidgets/trans_interface_kmeans.R")

# set some widths (doesn’t work if earlier)
svalue(trans_pane) <- 0.2

8.2.46 trans_interface_csp.R

# button pane
# csp_pane <- gpanedgroup(horizontal = TRUE,
#    expand = TRUE,
#    fill = TRUE,
#    container = trans_csp_tab)
#
## csp params
csp_param_frame <- gframe(text = "CSP Parameters",
    horizontal = FALSE,
    container = trans_csp_tab,
    expand = TRUE,
    width = 300)

## application params
csp_grouping_frame <- gframe(text = "Data Grouping",
    horizontal = FALSE,
    container = csp_param_frame,
    expand = TRUE,
    width = 300)

## opts
csp_grouping_layout <- glayout(container = csp_grouping_frame)
csp_avg_type_list <- c("Arithmetic", "Geometric", "Harmonic")
csp_grouping_layout[1,1] <- "Average Type"
csp_grouping_layout[2,1] <- gdroplist(csp_avg_type_list, text = "Average Type")
csp_grouping_layout[1,2] <- "Time Variable (Sample)"
csp_grouping_layout[2,2] <-
  gcombobox(
    names(rbci.env$importlist[[svalue(trans_var_filesel, index=TRUE)]]))
csp_grouping_layout[3,1] <- "Target Variable (Class)"
csp_grouping_layout[4,1] <-
  gcombobox(
    names(rbci.env$importlist[[svalue(trans_var_filesel, index=TRUE)]]))
csp_grouping_layout[3,2] <- "Channel Variable"
csp_grouping_layout[4,2] <-
  gcombobox(
    names(rbci.env$importlist[[svalue(trans_var_filesel, index=TRUE)]]))
csp_grouping_layout[5,1] <- "Epoch Group (Trial)"
csp_grouping_layout[6,1] <-
  gcombobox(
    names(rbci.env$importlist[[svalue(trans_var_filesel, index=TRUE)]]))
csp_grouping_layout[5,2] <- "Data Variable (Voltage)"
csp_grouping_layout[6,2] <-
  gcombobox(
    names(rbci.env$importlist[[svalue(trans_var_filesel, index=TRUE)]]))

addHandlerChanged(trans_var_filesel,
  handler = function(h,...) {
    new.dataset.names <-
      names(rbci.env$importlist[[svalue(trans_var_filesel, index=TRUE)]])
    csp_grouping_layout[2,2][] <- new.dataset.names
    csp_grouping_layout[4,1][] <- new.dataset.names
    csp_grouping_layout[4,2][] <- new.dataset.names
    csp_grouping_layout[6,1][] <- new.dataset.names
    csp_grouping_layout[6,2][] <- new.dataset.names
  })

### TODO make dynamic on dataset change
csp_feat_num_label <- glabel(text = "# of Spatial Pairs (0 = all)",
  container = csp_param_frame)
csp_feat_num <- gspinbutton(from = 0, to = 100, by = 1, 
                          container = csp_param_frame)

## output params

csp_output_frame <- gframe(text = "Output Options",
                           horizontal = FALSE, 
                           container = csp_param_frame, 
                           expand = TRUE, 
                           width = 300)

## extract csp button

csp_extract_btn <- gbutton("Extract CSP", 
                          container = csp_output_frame, 
                          handler = function (h,...) {
                            csp.name <- svalue(trans_var_filesel) 
                            csp.data <- rbci.env$importlist[[csp.name]] 
                            csp.timecol <- svalue(csp_grouping_layout[2,2]) 
                            csp.chancol <- svalue(csp_grouping_layout[4,2]) 
                            csp.valcol <- svalue(csp_grouping_layout[6,2]) 
                            csp.trialcol <- svalue(csp_grouping_layout[6,1]) 
                            csp.classcol <- svalue(csp_grouping_layout[4,1]) 
                            csp.avgtyp <- svalue(csp_grouping_layout[2,1]) 
                            csp.paircount <- svalue(csp_feat_num)

                            ## do csp calc, get model

                            new.table <- 
                              list(
                                transform.csp(table.data = csp.data, 
                                              time.col = csp.timecol, 
                                              chan.col = csp.chancol, 
                                              val.col = csp.valcol, 
                                              trial.col = csp.trialcol, 
                                              class.col = csp.classcol, 
                                              avg.type = csp.avgtyp, 
                                              pair.count = csp.paircount) 
                              )

                            names(new.table) <- paste(csp.name, 
                                                "csp", seq_along(new.table), 
                                                sep = ".")

                            rbci.env$importlist <- append(rbci.env$importlist, 
                                                          new.table)

                            ## ensure names are straight

                            names(rbci.env$importlist) <-
```r
make.unique(names(rbci.env$importlist))
```
```r
addHandlerClicked(csp_apply_button,
  handler = function(h,...) {
    new.datasets <-
      names(rbci.env$importlist)
    trans_var_filesel[] <- new.datasets
csp_apply_list[] <- new.datasets
  })

csp_apply_list <-
gdroplist(names(rbci.env$importlist),
        container = csp_output_frame)
```

---

### 8.2.47 trans_interface_kmeans.R

```r
# button pane
kmeans_pane <- gpanedgroup(horizontal = TRUE,
                           expand = TRUE,
                           fill = TRUE,
                           container = trans_kmeans_tab)

kmeans_varlist_frame <- gframe(text = "Cluster Column",
                                horizontal = FALSE,
                                container = kmeans_pane,
                                expand = TRUE,
                                width = 300)

# populate varlist
kmeans_varlist <- gcheckboxgroup(
  names(rbci.env$importlist[[svalue(trans_var_filesel, index=TRUE)]]),
  container = kmeans_varlist_frame,
  use.table = TRUE,
  expand = TRUE)

addHandlerChanged(trans_var_filesel,
  handler = function(h,...) {
    new.dataset.names <-
      names(rbci.env$importlist[[svalue(trans_var_filesel,
                                                 index=TRUE)]])
    kmeans_varlist[] <- new.dataset.names
  })

kmeans_action_pane <- gpanedgroup(horizontal = TRUE,
                                   expand = TRUE)
```

---

176
```r

## kmeans params
kmeans_param_frame <- gframe(text = "kmeans Parameters",
                              horizontal = FALSE,
                              container = kmeans_action_pane,
                              expand = TRUE,
                              width = 300)

## opts
kmeans_algorithm_type_list <- c("Hartigan-Wong", "Lloyd-Forgy", "MacQueen")
kmeans_algorithm_type_label <- glabel(text = "Algorithm",
                                       container = kmeans_param_frame)
kmeans_algorithm_type_menu <-
    gdroplist(kmeans_algorithm_type_list,
              text = "Algorithm",
              container = kmeans_param_frame)

# numerical entries (spinboxes)
kmeans_band_label <- glabel(text = "Numerical Parameters",
                             container = kmeans_param_frame)
kmeans_band_layout <- glayout(container = kmeans_param_frame)

kmeans_band_layout[1,1] <- "# of Centers"
kmeans_band_layout[2,1] <- gspinbutton(from = 0, by = 1)

kmeans_band_layout[1,2] <- "Max. Iterations"
kmeans_band_layout[2,2] <- gspinbutton(from = 1, by = 1)

## output params
kmeans_output_frame <- gframe(text = "Cluster Output Options",
                               horizontal = FALSE,
                               container = kmeans_param_frame,
                               expand = TRUE,
                               width = 300)

## apply kmeans button
kmeans_apply_btn <-
    gbutton("Cluster Data",
            container = kmeans_output_frame,
            handler = function (h,...) {
                k.type <- svalue(kmeans_algorithm_type_menu)
                k.center <- svalue(kmeans_band_layout[2,1])
            })
```
k.iter <- svalue(kmeans_band_layout[2,2])
k.dataname <- svalue(trans_var_filesel)
k.datafile <- rbci.env$importlist[[k.dataname]]
k.col <- svalue(kmeans_varlist)

## do clustering, get model
new.table <-
  list(
    transform.kmeans(k.datafile, k.col, 
      k.type, k.center, k.iter)
    #
    k.trial, k.chan)

names(new.table) <- paste(k.dataname,
  "kmeansmodel", seq_along(new.table),
  sep = ".")

rbci.env$importlist <- append(rbci.env$importlist,
  new.table)

names(rbci.env$importlist) <-
  make.unique(names(rbci.env$importlist))
}

## refresh dataset frame on run
addHandlerClicked(kmeans_apply_btn,
  handler = function(h,...){
    new.datasets <-
      names(rbci.env$importlist)
    trans_var_filesel[] <- new.datasets
ekmeans_data_list[] <- new.datasets
  })

## TODO alert complete

## plot variances
kmeans_plot_btn <-
gbutton("Plot Clustered Data",
  container = kmeans_output_frame,
  handler = function(h,...) {
    k.dataname <- svalue(trans_var_filesel)
k.datafile <- rbci.env$importlist[[k.dataname]]
k.tgtname <- svalue(kmeans_data_list)
k.tgtfile <- rbci.env$importlist[[k.tgtname]]

    print(clusplot(k.tgtfile,
      k.datafile$cluster,
      color=TRUE, shade=TRUE,
      k.cluster'
  })
## cluster plot target data

kmeans_plotdata_label <- glabel("Plot Data Set")
kmeans_data_list <-
gdroplist(names(rbci.env$importlist),
          container = kmeans_output_frame)

## plot pane

## kmeans plot on right side

kmeans_plot_frame <- ggraphics(container = kmeans_action_pane)

## set some widths (doesn’t work if earlier)

svalue(kmeansPane) <- 0.2
svalue(kmeans_action_pane) <- 0.2

---

8.2.48 trans_interface_pca.R

### button pane

pca_pane <- gpanedgroup(horizontal = TRUE,
                         expand = TRUE,
                         fill = TRUE,
                         container = trans_pca_tab)

### pca params

pca_param_frame <- gframe(text = "PCA Parameters",
                          horizontal = FALSE,
                          container = pca_pane,
                          expand = TRUE,
                          width = 300)

### opts

pca_kernel_type_list <- c("Linear","Gaussian","Laplace","Polynomial",
                          "Hyperbolic","Bessel","ANOVA")
pca_kernel_type_label <- glabel(text = "Kernel Type",
                                 container = pca_param_frame)
pca_kernel_type_menu <-
gdroplist(pca_kernel_type_list,
          text = "Kernel Type",
          container = pca_param_frame,
          handler = function (h,...) {
            ## enable or disable param GUI opts on type change
          })
## param names by type
## sigma for rbf/Laplace
## degree, scale, offset for Polynomial
## scale, offset for tanhdot
## sigma, order, degree, for Bessel
## sigma, degree for ANOVA

switch (svalue(h$obj),
    "Linear" = {
        enabled(pca_band_layout[1,1]) <- FALSE
        enabled(pca_band_layout[2,1]) <- FALSE
        enabled(pca_band_layout[1,2]) <- FALSE
        enabled(pca_band_layout[2,2]) <- FALSE
        enabled(pca_band_layout[3,1]) <- FALSE
        enabled(pca_band_layout[4,1]) <- FALSE
    },
    "Laplace" = {
        enabled(pca_band_layout[1,1]) <- TRUE
        enabled(pca_band_layout[2,1]) <- TRUE
        svalue(pca_band_layout[1,1]) <- "Sigma"
        enabled(pca_band_layout[1,2]) <- FALSE
        enabled(pca_band_layout[2,2]) <- FALSE
        enabled(pca_band_layout[3,1]) <- FALSE
        enabled(pca_band_layout[4,1]) <- FALSE
    },
    "Gaussian" = {
        enabled(pca_band_layout[1,1]) <- TRUE
        enabled(pca_band_layout[2,1]) <- TRUE
        svalue(pca_band_layout[1,1]) <- "Sigma"
        enabled(pca_band_layout[1,2]) <- FALSE
        enabled(pca_band_layout[2,2]) <- FALSE
        enabled(pca_band_layout[3,1]) <- FALSE
        enabled(pca_band_layout[4,1]) <- FALSE
    },
    "Polynomial" = {
        enabled(pca_band_layout[1,1]) <- TRUE
        enabled(pca_band_layout[2,1]) <- TRUE
        svalue(pca_band_layout[1,1]) <- "Degree"
        enabled(pca_band_layout[1,2]) <- TRUE
        enabled(pca_band_layout[2,2]) <- TRUE
        svalue(pca_band_layout[2,2]) <- "Scale"
        enabled(pca_band_layout[3,1]) <- TRUE
        enabled(pca_band_layout[4,1]) <- TRUE
        svalue(pca_band_layout[3,1]) <- "Offset"
    },
)
"Hyperbolic" = {
  enabled(pca_band_layout[1, 1]) <- TRUE
  enabled(pca_band_layout[2, 1]) <- TRUE
  svalue(pca_band_layout[1, 1]) <- "Scale"
  enabled(pca_band_layout[1, 2]) <- TRUE
  enabled(pca_band_layout[2, 2]) <- TRUE
  svalue(pca_band_layout[1, 2]) <- "Offset"
  enabled(pca_band_layout[3, 1]) <- FALSE
  enabled(pca_band_layout[4, 1]) <- FALSE
},
"Bessel" = {
  enabled(pca_band_layout[1, 1]) <- TRUE
  enabled(pca_band_layout[2, 1]) <- TRUE
  svalue(pca_band_layout[1, 1]) <- "Sigma"
  enabled(pca_band_layout[1, 2]) <- TRUE
  enabled(pca_band_layout[2, 2]) <- TRUE
  svalue(pca_band_layout[1, 2]) <- "Order"
  enabled(pca_band_layout[3, 1]) <- TRUE
  enabled(pca_band_layout[4, 1]) <- TRUE
  svalue(pca_band_layout[3, 1]) <- "Degree"
},
"ANOVA" = {
  enabled(pca_band_layout[1, 1]) <- TRUE
  enabled(pca_band_layout[2, 1]) <- TRUE
  svalue(pca_band_layout[1, 1]) <- "Sigma"
  enabled(pca_band_layout[1, 2]) <- TRUE
  enabled(pca_band_layout[2, 2]) <- TRUE
  svalue(pca_band_layout[1, 2]) <- "Degree"
  enabled(pca_band_layout[3, 1]) <- FALSE
  enabled(pca_band_layout[4, 1]) <- FALSE
}

# numerical entries (spinboxes)
pca_band_label <- glabel(text = "Numerical Parameters",
  container = pca_param_frame)
pca_band_layout <- glayout(container = pca_param_frame)

pca_band_layout[1, 1] <- ""
pca_band_layout[2, 1] <- gspinbutton(from = 0, to = 1, by = 0.01)
# stop band end
pca_band_layout[1,2] <- ""
pca_band_layout[2,2] <- gspinbutton(from = 0, to = 1, by = 0.01)

# pass band start
pca_band_layout[3,1] <- ""
pca_band_layout[4,1] <- gspinbutton(from = 0, to = 1, by = 0.01)

# pass band end
pca_band_layout[3,2] <- "# of Features (0 = all)"
pca_band_layout[4,2] <-
  gspinbutton(
    from = 0, # TODO fix upper limit
    to = 300,
    by = 1)

enabled(pca_band_layout[1,1]) <- FALSE
enabled(pca_band_layout[2,1]) <- FALSE
enabled(pca_band_layout[1,2]) <- FALSE
enabled(pca_band_layout[2,2]) <- FALSE
enabled(pca_band_layout[3,1]) <- FALSE
enabled(pca_band_layout[4,1]) <- FALSE

addSpring(pca_param_frame)

## application params
pca_grouping_frame <- gframe(text = "Data Grouping",
  horizontal = FALSE,
  container = pca_param_frame,
  expand = TRUE,
  width = 300)

# trial/group vars
pca_grouping_layout <- glayout(container = pca_grouping_frame)

pca_grouping_layout[1,1] <- "Data Variable (Voltage)"
pca_grouping_layout[2,1] <-
  gcombobox(
    names(rbci.env$importlist[[svalue(trans_var_filesel, index=TRUE)]]))

pca_grouping_layout[1,2] <- "Time Variable (Sample)"
pca_grouping_layout[2,2] <-
  gcombobox(
    names(rbci.env$importlist[[svalue(trans_var_filesel, index=TRUE)]]))

pca_grouping_layout[3,1] <- "Target Variable (Class)"
pca_grouping_layout[4,1] <-
  gcombobox(
    names(rbci.env$importlist[[svalue(trans_var_filesel, index=TRUE)]]))
pca_grouping_layout[3,2] <- "Channel Variable"
pca_grouping_layout[4,2] <-
gcombobox(
    names(rbci.env$importlist[[svalue(trans_var_filesel, index=TRUE)]])
)
pca_grouping_layout[5,1] <- "Epoch Group (Trial)"
pca_grouping_layout[6,1] <-
gcombobox(
    names(rbci.env$importlist[[svalue(trans_var_filesel, index=TRUE)]])
)

## change column selectors on dataset change
addHandlerChanged(trans_var_filesel,
    handler = function(h,...) {
        new.dataset.names <-
            names(rbci.env$importlist[[svalue(trans_var_filesel, index=TRUE)]])
        pca_grouping_layout[2,1][] <- new.dataset.names
        pca_grouping_layout[2,2][] <- new.dataset.names
        pca_grouping_layout[4,1][] <- new.dataset.names
        pca_grouping_layout[4,2][] <- new.dataset.names
        pca_grouping_layout[6,1][] <- new.dataset.names
    })

### output params
pca_output_frame <- gframe(text = "PCA Output Options",
    horizontal = FALSE,
    container = pca_param_frame,
    width = 300)
pca_output_layout <- glayout(container = pca_output_frame,
    expand = TRUE)

## compute pca button
pca_output_layout[1,1] <-
gbutton("Compute PCA",
    container = pca_output_frame,
    handler = function(h,...){
        ## collect args
        input.name <- svalue(trans_var_filesel)
        pca.args <- list(
            input.table = bquote( # partial deref
                rbci.env$importlist[[.(input.name)]]),
            val.col = svalue(pca_grouping_layout[2,1]),
            targ.name = svalue(pca_grouping_layout[4,1]),
            epoch.name = svalue(pca_grouping_layout[6,1]),
            time.name = svalue(pca_grouping_layout[2,2]),
        )
    })
split.col = svalue(pca_grouping_layout[4,2]),
kernel.type = svalue(pca_kernel_type_menu),
pc.count = svalue(pca_band_layout[4,2])
)

new.table <-
  list( do.call(transform.pca, pca.args) )

names(new.table) <- paste(input.name,
    "pcamodel", seq_along(new.table),
    sep = ".")

rbci.env$importlist <- append(rbci.env$importlist,
  new.table)

## ensure names are straight
names(rbci.env$importlist) <- make.unique(names(rbci.env$importlist))

## add op to reporter
add.step("transform.pca", pca.args)
}

addHandlerClicked(pca_output_layout[1,1],
  handler = function(h,...){
    new.datasets <-
      names(rbci.env$importlist)
    trans_var_filesel[] <- new.datasets
    pca_output_layout[3,1][] <- new.datasets
  })

# alert complete (progress bar?)

# plot variances
pca_output_layout[1,2] <-
gbutton("Plot Eigenvalues",
  container = pca_output_frame,
  handler = function(h,...){
    plot.args <- list(
      bquote(rbci.env$importlist[[.(svalue(trans_var_filesel))]])
    )
    print(do.call(plot,plot.args))
    ## add op to reporter
    add.step("plot", plot.args)
  })

  pca_output_layout[2,2] <-
gbutter("Plot 2D Subspaces",
## container = pca_output_frame,
handler = function(h,...){
## TODO add hex binning etc.
## see https://github.com/vqv/ggbiplot/blob/master/README.markdown
  pca.plotargs <- list(
    pcobj = bquote(
      rbci.env$importlist[[.(svalue(trans_var_filesel))]]
    )
  )

  print(do.call(ggbiplot, pca.plotargs))
  handler = function(h,...){
    data.name <- svalue(pca_output_layout[3,1])
    pca.name <- svalue(trans_var_filesel)
    pc.args <- list(
      long.data.set = bquote(# partial deref
        rbci.env$importlist[[.(data.name)])],
      pca.model = bquote(# partial deref
        rbci.env$importlist[[.(pca.name)])],
      val.col = svalue(pca_grouping_layout[2,1]),
      targ.name = svalue(pca_grouping_layout[4,1]),
      epoch.name = svalue(pca_grouping_layout[6,1]),
      time.name = svalue(pca_grouping_layout[2,2]),
      split.col = svalue(pca_grouping_layout[4,2])
    )

    new.table <- list(do.call(transform.pc, pc.args)) # do transform
    add.step("transform.pc", pc.args) # add op to reporter

    ## add to dataset list
    names(new.table) <- paste(pca.name,
      "pc", seq_along(new.table),
      sep = ".")
    rbci.env$importlist <- append(rbci.env$importlist,
      names(new.table) <-
      make.unique(names(rbci.env$importlist))
    })
  })
)}

addHandlerClicked(pca_output_layout[2,1],}
handler = function(h,...){
  new.datasets <-
    names(rbci.env$importlist)
  trans_var_filesel[] <- new.datasets
  pca_output_layout[3,1][] <- new.datasets
}

pca_output_layout[3,1] <-
  gdroplist(names(rbci.env$importlist))
#  container = pca_output_frame)
## we ALSO want to have column selector change when selecting target sets for
## the target set list
addHandlerChanged(pca_output_layout[3,1],
  handler = function(h,...) {
    new.dataset.names <-
      names(rbci.env$importlist[[svalue(pca_output_layout[3,1])]])
    pca_grouping_layout[2,1][] <- new.dataset.names
    pca_grouping_layout[2,2][] <- new.dataset.names
    pca_grouping_layout[4,1][] <- new.dataset.names
    pca_grouping_layout[4,2][] <- new.dataset.names
    pca_grouping_layout[6,1][] <- new.dataset.names
  })

# plot pane
# pca plot on right side
pca_plot_frame <- ggraphics(container = pca_pane)
# set some widths (doesn’t work if earlier)
svalue(pca_pane) <- 0.4
Appendix: List of Acronyms

- API: Application programming interface
- AUTD: Airborne ultrasonic tactile display
- BCI: Brain-computer interface
- CPAN: Community Perl Archive Network
- CPU: Central processing unit
- CRAN: Community R Archive Network
- CSP: Common spatial pattern
- CSV: Comma-separated values
- CTAN: Community TeX Archive Network
- DARPA: Defense Advanced Research Projects Agency
- DDR: Double data rate
- DIMM: Dual in-line memory module
- EDF: European data format
- EEG: Electroencephalogram
- ERD: Event-related desynchronization
- ERP: Event-related potential
- FBI: Federal Bureau of Investigation
- FDA: Food and Drug Association
- GIMP: Gnu Image Manipulator
- GNU: GNU’s not Unix! (The free software packages that support a Linux/UNIX kernel)
- GPLv3: GNU Public License version 3
- GTK: GIMP Toolkit
- GUI: Graphical user interface
- HPC: High-performance computing
- ISO: International standards organization
- LDA: Linear discriminant analysis
- LED: Light-emitting diode
- doMC: do-Multicore
- MEG: Magnetoencephalogram
- MPI: Message passing interface
• MVAR: Modeling of variance
• OS: Operating system
• PCA: Principal component analysis
• SBIR: Small Business Innovation Research
• SDA: Shrinkage discriminant analysis
• SIFT: Source Information Flow Toolbox
• SNOW: Simple network of workstations
• SSH: Secure shell server
• SSVEP: Steady-state visually evoked potential
• SVM: Support vector machine
• SWLDA: Stepwise linear discriminant analysis
• TSV: Tab-separated values
• UDP: User datagram protocol
10 References


g.tec medical engineering GmbH. Products / Home. 2014. URL: http://www.gtec.at/Products/(l)71.


LF Haas. “Hans Berger (18731941), Richard Caton (18421926), and electroencephalography”. In: Journal of Neurology, Neurosurgery & Psychiatry 74.1 (2003), p. 9. DOI: 10.1136/jnnp.74.1.9. eprint: http://jnnp.bmj.com/content/74/1/9.full.pdf+html. URL: http://jnnp.bmj.com/content/74/1/9.short.


Hadley Wickham. _lineprof: An alternative display for line profiling information_. R package version 0.1.


