
MATCLASSRSA v2

USER MANUAL

BERNARD C. WANG, RAYMOND GIFFORD, NATHAN C. L. KONG, FENG RUAN,
ANTHONY M. NORCIA, AND BLAIR KANESHIRO
DECEMBER 2025

Table of Contents

1	Introduction	4
1.1	Toolbox Overview	4
1.2	Toolbox Background and History	7
1.3	Citing the Toolbox	8
1.4	License	9
1.5	Support	10
1.6	Acknowledgments	10
1.7	Author Information and Contributions	10
1.8	Declaration on the Usage of AI	11
2	Getting Started	12
2.1	Toolbox and Resources	12
2.1.1	GitHub Repository	12
2.1.2	User Manual and Preprint	12
2.1.3	Example Data	13
2.2	Setup and Installation	13
2.2.1	Operating Systems and MATLAB versions	13
2.2.2	Dependencies	13
2.2.3	Installation	14
2.3	Using the Toolbox	15
2.3.1	Function-Based Toolbox	15
2.3.2	Toolbox Folder Structure	15
2.3.3	Modules and Main Function Calls	16
2.3.4	Helper Functions	17
2.3.5	Example Code and Data	17
2.4	Input and Output Specifications	18
2.4.1	Input Data Specifications	18
2.4.2	Function Outputs	20
3	Module: Preprocessing	21
3.1	Overview	21
3.2	shuffleData()	22
3.3	averageTrials()	24
3.4	noiseNormalization()	27
4	Module: Reliability (Data Quality Assessments)	29
4.1	Overview	29
4.2	computeSpaceTimeReliability()	30
4.3	computeSampleSizeReliability()	33

5	Module: Classification	36
5.1	Overview	36
5.2	High-level Overview: M/EEG Classification for RSA	37
5.3	Overview of MatClassRSA Classification Specifications	39
5.3.1	Core inputs: Data Matrix and Labels Vector	39
5.3.2	Subsetting Data Features	39
5.3.3	Classifier Selection	39
5.3.4	Classifier Optimization	40
5.3.5	Principal Components Analysis (PCA)	40
5.3.6	Data Centering and Scaling	41
5.3.7	Cross Validation	41
5.3.8	Permutation testing	42
5.3.9	Classifier outputs	44
5.4	<code>crossValidateMulti()</code>	44
5.5	<code>crossValidateMulti_opt()</code>	50
5.6	<code>crossValidatePairs()</code>	55
5.7	<code>crossValidatePairs_opt()</code>	61
5.8	<code>trainMulti()</code>	67
5.9	<code>trainMulti_opt()</code>	72
5.10	<code>trainPairs()</code>	76
5.11	<code>trainPairs_opt()</code>	81
5.12	<code>predict()</code>	87
6	Module: RDM Computation	90
6.1	Overview	90
6.2	<code>computeCMRDM()</code>	91
6.3	<code>shiftPairwiseAccuracyRDM()</code>	94
6.4	<code>computeEuclideanRDM()</code>	95
6.5	<code>computePearsonRDM()</code>	97
7	Module: Visualization	100
7.1	Overview	100
7.2	Example Figure Code	101
7.3	<code>plotMatrix()</code>	102
7.4	<code>plotMDS()</code>	105
7.5	<code>plotDendrogram()</code>	107
7.6	<code>plotMST()</code>	111
8	Illustrative Examples	114
8.1	Illustrative 0: Example data download	114

8.2	Illustrative 1: Downstream impacts of preprocessing	117
8.3	Illustrative 2: Single-channel analyses	128
8.4	Illustrative 3: Time-resolved analyses	135
8.5	Illustrative 4: Train-test SVM optimization	144
8.6	Illustrative 5: Compare different RDM constructions	147
8.7	Illustrative 6: Customizing figures with stimulus images	156
9	Appendix: Helper functions	160
10	References	176

Introduction

1.1 Toolbox Overview

Classification of magnetoencephalography (MEG) and electroencephalography (EEG) is a brain decoding technique (Holdgraf et al., 2017) in which the goal is to correctly predict labels of test observations based on a statistical model built from labeled training observations (Hastie et al., 2009). EEG classification has origins in the brain-computer interface (BCI) domain, where the goal is to enable users to communicate or perform actions through brain activity alone (Blankertz et al., 2002). Whether decoding motor imagery (Pfurtscheller and Neuper, 2001), steady-state evoked responses (Allison et al., 2008; Kim et al., 2011), or event-related potentials (ERPs) (Farwell and Donchin, 1988; Halder et al., 2010), BCI design is centered around achieving accurate, real-time decoding.

M/EEG classification is used in cognitive neuroscience as well. While a well-performing classifier remains integral to these investigations, the more granular classifier outputs—namely, the degree of separability or confusion among responses to various stimuli—provides new avenues through which to investigate the neural processes underlying categories of stimuli, brain states, and participant attributes. Reports of classifier success (i.e., accuracies) and confusion (i.e., misclassifications) over stimulus sets serve to convey the distances or similarities, respectively, across the set.

This perspective of classifier outputs is aligned with the Representational Similarity Analysis (RSA) paradigm. In RSA, a set of stimulus representations from any data modality—including M/EEG data—can be abstracted to a Representational Dissimilarity Matrix (RDM), which summarizes the pairwise distances between all of the stimuli (Kriegeskorte et al., 2008a; Kriegeskorte et al., 2008b). Through RDMs, data from diverse modalities are brought into a common ‘representational space’, and the similarity between RDMs can be quantitatively assessed to determine, for example, which computational or conceptual representation of a stimulus set aligns with which neural representation. Moreover, since M/EEG data are multidimensional in both time and space, RDMs can be constructed on data subsets in a ‘searchlight’ approach to identify specific spatiotemporal features that best align with other RDMs (Su et al., 2014; Cichy et al., 2014; Carlson et al., 2013; Kong et al., 2020). Moreover, RDMs can undergo hierarchical and non-hierarchical clustering to create visualizations that convey the distances among the stimuli in various ways (Carlson et al., 2013; Kaneshiro et al., 2015b; Losorelli et al., 2020).

To date, M/EEG classification has been used to investigate multiple topics including object category representation (C. Wang et al., 2012; Carlson et al., 2013; Cichy et al., 2014; Kaneshiro et al., 2015b; Kong et al., 2020; Xie et al., 2022), language categories (Suppes et al., 2009), visual and auditory representation of stimuli (Simanova et al., 2010), auditory attention (An et al., 2023), development (Xie et al., 2022) and correspondences to RDMs derived from other data types including fMRI (Cichy et al., 2014; Muukkonen et al., 2020) and conceptual or computational stimulus models (Wardle et al., 2016; Kong et al., 2020; Li et al., 2022). While most M/EEG RSA classification studies involve ERPs and adult participants, the approach has also been shown to work in Rapid Serial Visual Presentation (RSVP) paradigms (Grootswagers et al., 2019), on auditory frequency-following responses (Losorelli et al., 2020), on EEG alpha power (An et al., 2023) representations, and on infant data (Xie et al., 2022).

With the aim of facilitating cognitive neuroscience research for E/MEG researchers who are interested in classification and RSA, we present the v2 release of MatClassRSA. MatClassRSA is a MATLAB toolbox that performs M/EEG classification and other analyses related to constructing RDMs for RSA. As noted below, the toolbox is organized in a modular toolbox structure, where outputs of some modules can feed into others. In addition to classifying data and computing RDMs, MatClassRSA includes functions for assessing data quality; optionally preprocessing the data; and visualizing RDMs. This release is a major update and expansion of the original toolbox (B. C. Wang et al., 2017). Specifically, the v2 release includes additional functionalities (preprocessing, reliability, and RDM analyses of sensor-space data); a restructuring of the codebase; expanded documentation and design rationales in the User Manual; scripts offering example function calls; and new illustrative analyses.

Target end users of the toolbox are those who already work with repeated-trials M/EEG data in MATLAB (for e.g., ERP analyses) and wish to perform new types of analyses with these data. By removing the burden of implementing standard machine-learning and other analysis procedures by hand, we hope that the toolbox makes these computationally focused analyses accessible to a wider range of researchers.

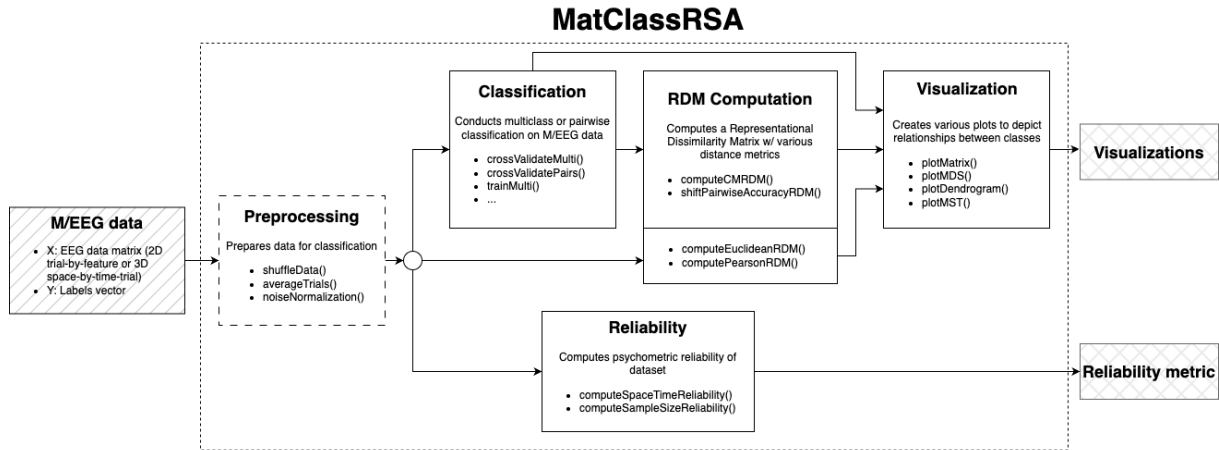


Figure 1: **Overview of MatClassRSA modules and main functions.**

MatClassRSA provides multiple functionalities directly and tangentially related to M/EEG classification for RSA. MatClassRSA comprises five modules, as summarized in the white boxes of Figure 1:

1. **Preprocessing:** While MatClassRSA does not offer basic M/EEG data cleaning such as filtering and epoching, this module includes additional preprocessing steps to apply to already-cleaned data prior to downstream analyses. For instance, users can shuffle the ordering of the trials, average groups of single trials to improve SNR, and normalize the data based on each sensor's SNR.
2. **Reliability:** Users may wish to assess the reliability of their data to obtain an upper bound on the data's explainability by any model. This module contains functions to do so across each spatial and/or temporal feature, as well as across varying sample sizes of the data.
3. **Classification:** This module includes all of the classification functionalities. Separate functions handle non-optimized and optimized classifications in multiclass and pairwise contexts. Users can iteratively train and test using cross-validation functions, or call separate train and predict functions.
4. **RDM Computation:** This module includes functions to convert classifier outputs (i.e., multiclass confusion matrices, pairwise accuracy matrices) to RDMs. It also includes

two non-classification RDM functions that operate directly on the input data, based on Euclidean distance and Pearson correlation.

5. **Visualization:** The final module includes functions for visualizing proximity matrices as well as multidimensional scaling (MDS), dendrogram, and minimum spanning tree (MST) representations of a proximity matrix.

The dashed lines around the Preprocessing module in Figure 1 indicate that M/EEG data may be *optionally* run through functions in the Preprocessing module before being input to the Reliability, Classification, and/or RDM Computation modules, or may be directly input to the latter modules. As noted on the right side of the figure, Reliability functions output reliability metrics, while Classification and RDM Computation outputs can ultimately be visualized via the Visualization module.

We note that MatClassRSA emphasizes the *construction* of RDMs but does not perform downstream RSA analyses, such as quantitative comparisons across RDMs. For these analyses, users are encouraged to consult other existing toolboxes such as the MATLAB RSA toolbox by Nili et al. (2014).

MatClassRSA v2 joins an ecosystem of open pipelines and toolboxes centered around M/EEG classification. There has been remarkable growth in this area in recent years, expanding upon BCI-focused resources (Blankertz et al., 2011; Kothe and Makeig, 2013) to include a number of MATLAB toolboxes that collectively offer classification as well as related functionalities including advanced statistics (Oosterhof et al., 2016; Fahrenfort et al., 2018; López-García et al., 2022), emphasis on figure generation (Fahrenfort et al., 2018), and graphical user interfaces (GUIs) (Ghorbani et al., 2020; López-García et al., 2022). These resources complement decoding-focused tutorials (Grootswagers et al., 2017; Guggenmos et al., 2018), Python toolboxes (Hanke et al., 2009; Krell et al., 2013; Appriou et al., 2021; Kuntzelman et al., 2021; Jana et al., 2025), toolboxes for other imaging modalities (Schrouff et al., 2013; Grotegerd et al., 2014), and more general M/EEG analysis toolboxes (Delorme and Makeig, 2004; Oostenveld et al., 2011; Gramfort et al., 2013; Gramfort et al., 2014). Among all of these options, MatClassRSA may be most useful for researchers who are interested in classification and obtaining basic statistics, non-classification-based reliability and RDM operations, and clustering and visualization of RDMs.

1.2 Toolbox Background and History

This software was originally developed for members of the Stanford Vision and Neuro-Development Lab (SVNDL), Stanford Translational Auditory Research (STAR) Lab, and Music Engagement Research Initiative (MERI) to perform EEG classification in conjunction with visualizations related to Representational Similarity Analysis. Based on the level

of interest in the software, we eventually packaged the functionalities into a toolbox for greater benefit to the general research community. The initial version of MatClassRSA was released as B. C. Wang et al. (2017). The current v2 release represents a major revision of the toolbox, including updated package folder structure, new Preprocessing and Reliability modules, expansion of the Classification and RDM Computation modules, and improved methodologies.

Several aspects of MatClassRSA v2 draw from previously published research, code, and data:

- Across both MatClassRSA versions, many functions were designed around the analyses reported in Kaneshiro et al. (2015b).
- As detailed in this User Manual and the MatClassRSA v2 preprint (B. C. Wang et al., 2025b), certain functions in the current release were adapted from or make direct use of code reported in Guggenmos et al. (2018) and their related tutorials.¹ Their usage is noted in both the function documentation in this manual as well as in respective MatClassRSA function docstrings.
- Some data files in the accompanying MatClassRSA v2 dataset (B. C. Wang et al., 2025a) are drawn, with attribution, from previously published datasets and publications (Kaneshiro et al., 2015a; Kong et al., 2020; Losorelli et al., 2019).

Finally, previous versions of MatClassRSA functions have been used by members of our author team in peer-reviewed publications (Kong et al., 2020; Losorelli et al., 2020).

1.3 Citing the Toolbox

The MatClassRSA v2 release comprises three deliverables: The codebase, a preprint summarizing the release (B. C. Wang et al., 2025b), and a dataset containing example data for the illustrative analyses (B. C. Wang et al., 2025a).

If using any code from the MatClassRSA v2 release, please cite the following two items:

- **v2 release preprint**

Bernard C. Wang, Raymond Gifford, Nathan C. L. Kong, Feng Ruan, Anthony M. Norcia, and Blair Kaneshiro (2025). MatClassRSA v2 Release: A MATLAB Toolbox for M/EEG Classification, Proximity Matrix Construction, and Visualization. bioRxiv 2025.11.19.689115. doi:10.1101/2025.11.19.689115

¹https://github.com/m-guggenmos/megmvpa/blob/master/tutorial_matlab/matlab_distance.ipynb

- **GitHub repository**

Please cite the Zenodo record and DOI for the most recent GitHub release, as referenced in the bibliography of the above preprint.

If using any of the toolbox's example data in outside projects, please also cite the following:

- **MatClassRSA dataset**

Bernard C. Wang, Raymond Gifford, Nathan C. L. Kong, Feng Ruan, Anthony M. Norcia, and Blair Kaneshiro (2025). Example Data for MatClassRSA v2 Release. Stanford Digital Repository. Available at <https://purl.stanford.edu/kv831rr3606/>. doi:10.25740/kv831rr3606.

1.4 License

MatClassRSA v2 is released under the MIT License,² as follows:

Copyright (c) 2025 Bernard C. Wang, Raymond Gifford, Nathan C. L. Kong, Feng Ruan, Anthony M. Norcia, and Blair Kaneshiro.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

If a code file within the toolbox originated from outside of MatClassRSA and already contained a license, that original license is retained.

²<https://choosealicense.com/licenses/mit/>

1.5 Support

For questions, comments, suggestions, feature requests, and bug reports, contact Blair Kaneshiro <blairbo@ccrma.stanford.edu> and Bernard Wang <bernardcwang@gmail.com>.

1.6 Acknowledgments

This research was supported by the Patrick Suppes Gift Fund and the Roberta Bowman Denning Fund for Humanities and Technology (BK). The authors gratefully acknowledge Yiran Duan, Steven Losorelli, Vaidehi Natsu, Duc T. Nguyen, Ethan Roy, Greta Vilidaite, and Xiaoqian Yan for their valuable feedback and insights contributed while using development versions of the toolbox. We also thank Peter J. Kohler for technical suggestions, Daniel O’Leary and Andero Uusberg for advice on input data specifications, Amilcar Malave for guidance on the code release, and an anonymous reviewer for constructive feedback on a previous version of the toolbox. Finally, we thank Marcos Perreau Guimaraes and Hyung-Suk Kim for their past contributions that are reconceptualized in this toolbox; and Patrick Suppes for championing M/EEG classification in cognitive neuroscience research.

This User Manual was prepared using the *Extensive L^AT_EX Guide* Overleaf template by Armin Dubert.³

1.7 Author Information and Contributions

Authors of MatClassRSA v2:

- **Bernard C. Wang** – Center for Computer Research in Music and Acoustics, Stanford University, Stanford, CA 94305, USA
- **Raymond Gifford** – Center for Computer Research in Music and Acoustics, Stanford University, Stanford, CA 94305, USA
- **Nathan C. L. Kong** – Department of Psychology, Stanford University, Stanford, CA 94305, USA
- **Feng Ruan** – Department of Statistics, Stanford University, Stanford, CA 94305, USA
- **Anthony M. Norcia** – Department of Psychology and Wu Tsai Neurosciences Institute, Stanford University, Stanford, CA 94305, USA
- **Blair Kaneshiro** – Center for Computer Research in Music and Acoustics, Center for the Study of Language and Information, and Graduate School of Education, Stanford University, Stanford, CA 94305, USA

All authors approve the v2 release of MatClassRSA. The authors contributed as follows:

³<https://www.overleaf.com/articles/extensive-latex-guide/vdgrdqdwjhtk>

- Designed the toolbox architecture: BCW, AMN, BK
- Implemented the code: BCW, RG, NCLK, BK
- Validated the code: BCW, RG, NCLK, BK
- Documented the toolbox: BCW, RG, NCLK, BK
- Created illustrative analyses: RG, BK
- Provided theoretical and domain expertise: FR, AMN
- Wrote the preprint: BK
- Provided feedback and editing: BCW, RG, NCLK, FR, AMN, BK

1.8 Declaration on the Usage of AI

The authors used Azure OpenAI gpt-4o-mini through Stanford University's AI Playground platform⁴ for assistance in performing Git and \LaTeX operations as well as for clarifying, debugging, and documenting code files written by co-authors who had become less active on the project. We did not otherwise use any AI tools for developing the code or for writing the user manual or preprint.

⁴<https://uit.stanford.edu/service/aiplayground>

Getting Started

2.1 Toolbox and Resources

2.1.1 GitHub Repository

The MatClassRSA toolbox is available as a publicly accessible GitHub repository.⁵ The most recent v2.* GitHub release is the version associated with this documentation. The citation for the repository is associated with the linked instance on Zenodo (see Chapter 1.3 for more information). As noted in Chapter 1.4, MatClassRSA is published under an MIT license.

2.1.2 User Manual and Preprint

The MatClassRSA v2 release includes two forms of written documentation.

First, the User Manual provided in the GitHub repository (this document) is intended to provide comprehensive documentation of MatClassRSA, including brief background on M/EEG classification; guidance on getting up and running with the toolbox; documentation of the main and helper function; and walkthroughs of the illustrative analyses. This User Manual also elaborates on the documentation of the classification functions to provide more context on that module's general architecture and design decisions.

⁵<https://github.com/berneezy3/MatClassRSA>

Next, the preprint of the v2 release is available on bioRxiv (B. C. Wang et al., 2025b). It provides a more narrative overview of the toolbox, including its structure, functions, and illustrative analyses.

2.1.3 Example Data

Due to file size, example data files used in the illustrative analyses are not included in the MatClassRSA repository on GitHub and hence this folder is initially empty. They are instead provided as a separate dataset through the Stanford Digital Repository (SDR) (B. C. Wang et al., 2025a). The first script in the IllustrativeAnalyses folder (`illustrative_0_downloadExampleData.m`) downloads the data files from the SDR URLs into the ExampleData folder in the user's local instance of MatClassRSA. More information on the example data files is provided in the README of the online dataset; more information on the downloading procedure is provided in the Illustrative Analyses documentation (Chapter 8 of this manual).

2.2 Setup and Installation

2.2.1 Operating Systems and MATLAB versions

MatClassRSA v2 was tested on the following MacOS operating systems: Monterey 12.* and Sequoia 15.*. The original release of MatClassRSA was validated on both Mac and Linux operating systems, and we anticipate that the current release will similarly work across operating systems, but this has not been comprehensively validated.

The v2 release was developed and tested on recent versions of MATLAB including R2021a, R2024b, and R2025b. The code was developed with backward compatibility in mind, and functions involving MATLAB input parser⁶ and parallel pool⁷ functionalities have been written to accommodate older (e.g., 2016b) as well as current syntaxes. The toolbox may work with earlier versions of MATLAB, but the toolbox has not been fully tested on previous versions. MatClassRSA users can use MATLAB's code compatibility reporting tool⁸ to identify any issues in running the toolbox with their version of MATLAB.

2.2.2 Dependencies

The software requires two MATLAB toolboxes: The Parallel Computing Toolbox⁹ and the Statistics and Machine Learning Toolbox.¹⁰ Some of the MatClassRSA illustrative analyses

⁶<https://www.mathworks.com/help/matlab/ref/inputparser.html>

⁷<https://www.mathworks.com/help/parallel-computing/run-code-on-parallel-pools.html>

⁸<https://www.mathworks.com/help/matlab/ref/codecompatibilityreport.html>

⁹<https://www.mathworks.com/products/parallel-computing.html>

¹⁰<https://www.mathworks.com/products/statistics.html>

additionally require the Image Processing Toolbox¹¹.

The sole external dependency is LIBSVM, which is included in the `src` folder of the MatClassRSA GitHub repository:

Chang, Chih-Chung and Lin, Chih-Jen (2011). LIBSVM: A library for support vector machines. ACM transactions on intelligent systems and technology (TIST), 2(3), 1-27. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.

2.2.3 Installation

Clone or download the latest version of MatClassRSA at <https://github.com/berneezy3/MatClassRSA/>

The external dependency, LIBSVM, must be set up before any SVM classifications can be performed. Once inside the MatClassRSA main directory, navigate to

```
src/libsvm-3.21/matlab
```

and refer to the README file there for LIBSVM installation instructions.

LIBSVM note for Mac/Xcode users: Users running the `make.m` file in the `matlab` folder of LIBSVM may encounter the error shown in Figure 2. A post in MATLAB's online forum¹² suggests that this is due to an issue in recent versions of Xcode (e.g., versions 26.*) and offers a workaround. Alternatively, users can revert to a previous version of Xcode.¹³ As of the v2 release of MatClassRSA, we encountered issues running the makefile with Xcode versions 26.0 and 26.0.1; we were not able to implement the suggested workaround but were able to run `make.m` successfully after reverting Xcode to version 16.4.

After setting up LIBSVM, the user can run the following in the MATLAB command window to add the entire MatClassRSA toolbox to their path:

```
> MatClassRSAPath = 'path/to/your/directory/in/char/format';  
> addpath(genpath(MatClassRSAPath));
```

All MatClassRSA functions will now be runnable. To automatically import MatClassRSA upon MATLAB startup, create a script titled `startup.m` somewhere in the MATLAB search path (this can be found using the `path` command), and add the above lines into it.

¹¹<https://www.mathworks.com/products/image-processing.html>

¹²<https://www.mathworks.com/matlabcentral/answers/2180302-mex-failing-to-compile-function>

¹³<https://xcodereleases.com/>

```
Error: /Users/[redacted]/MatClassRSA/src/libsvm-3.24/matlab/make.m failed (line 15)
Undefined symbols for architecture arm64:
  "_mexCreateMexFunction", referenced from:
      <initial-undefines>
  "_mexDestroyMexFunction", referenced from:
      <initial-undefines>
  "_mexFunctionAdapter", referenced from:
      <initial-undefines>
ld: symbol(s) not found for architecture arm64
clang++. error: linker command failed with exit code 1 (use -v to see invocation)

=> Please check README for detailed instructions.
```

Figure 2: Potential LIBSVM makefile error related to Xcode on Mac.

2.3 Using the Toolbox

2.3.1 Function-Based Toolbox

MatClassRSA is a collection of functions and does not have a graphical user interface (GUI). Users should be comfortable writing scripts and functions or working in the MATLAB command line to load data, save outputs, and call MatClassRSA functions.

To help users get up and running with MatClassRSA, the following resources are provided throughout the toolbox:

- Each of the main user-called functions contains a detailed docstring which can be viewed in the respective code file or by accessing the function help by typing, e.g.,
`> help functionName`
in the MATLAB command line.
- For each of the main user-called functions, an elaborated description and example function calls are also provided in the User Manual (this document);
- Helper functions contain shorter docstrings in the code file and brief descriptions in the User Manual;
- Scripts containing example function calls for each of the main user-called functions are provided in the `ExampleFunctionCalls` folder;
- Illustrative analyses demonstrating advanced usage of the toolbox are provided in the `IllustrativeAnalyses` folder.

2.3.2 Toolbox Folder Structure

The top level of the MatClassRSA repository contains the following folders and files:

- ***ExampleData*** (folder): Storage location for data files used in the example function calls and illustrative analyses. Data files are not provided as part of the codebase but can be downloaded using code provided in the `IllustrativeAnalyses` folder. See the Illustrative Analyses documentation (Chapter 8 of this manual) for more information on the example data files and downloading procedure.
- ***ExampleFunctionCalls*** (folder): Location of scripts, each containing example function calls for one of the main user-called functions of the toolbox. As noted above, users need to have downloaded the example data into the `ExampleData` folder in order to run the example scripts.
- ***IllustrativeAnalyses*** (folder): Location of illustrative analyses demonstrating advanced usage of MatClassRSA. Users again need to have downloaded the example data into the `ExampleData` folder to run these analyses.
- ***src*** (folder): Main folder of MatClassRSA functions. This folder contains the primary MatClassRSA toolbox modules, namely the `+Preprocessing`, `+Reliability`, `+Classification`, `+RDM_Computation`, and `+Visualization` folders. It also includes a folder called `libsvm-3.24`, which contains the LIBSVM installation needed for selected classification calculations.
- ***.gitignore*** (file): Specification of files to remain untracked by Git, including the example data files stored in the `ExampleData` folder or elsewhere in the repository.
- ***LICENSE.md*** (file): License for the repository (MIT license).
- ***MatClassRSA v2 User Manual.pdf*** (file — this file): User Manual for the current release. Includes M/EEG classification and toolbox background; guidance for getting started with the toolbox; comprehensive documentation of user-called functions; supplemental overview and design rationales for the Classification module; description and results of illustrative analyses; brief documentation of helper functions.
- ***README.md*** (file): The README for the GitHub repository.

2.3.3 Modules and Main Function Calls

The main modules and functions of MatClassRSA are summarized in Figure 1 of the previous chapter. Each module is a folder inside of the `src` folder in the GitHub repository, and each function is a file inside of the respective module folder. The functions listed in the figure collectively represent the main user-called functions of the toolbox and receive the most attention in terms of documentation and examples.

As shown in the GitHub repository, the MatClassRSA module folder names begin with '+'. These denote MATLAB package folders. Accordingly, the main MatClassRSA functions are called by specifying the package folder name (without the '+'), followed by the function name. For example the `noiseNormalization()` function in the `+Preprocessing` folder can be called as follows:¹⁴

```
[normData, sigmaInv] = Preprocessing.noiseNormalization(X, Y);
```

2.3.4 Helper Functions

Many of the main MatClassRSA functions call helper functions, which are all located in the `+Utils` folder of the `src` folder. Users need not interact directly with the helper functions in order to make use of the main MatClassRSA functions. However, the helper functions are documented—albeit in a more cursory fashion—in their docstrings and in this User Manual (Chapter 9) for users who wish to use or extend them for custom analyses.

2.3.5 Example Code and Data

MatClassRSA comes with example function calls for each main user-called function in the `ExampleFunctionCalls` folder, as well as a collection of illustrative examples in the `IllustrativeAnalyses` folder. The example function calls demonstrate various ways to call each function with randomly generated data, while the illustrative analyses demonstrate more in-depth use cases—such as multi-function analyses and illustrations of how preprocessing or classification decisions can impact downstream results—with real data. The example function calls are referenced in each function's documentation, while the illustrative analyses are described in Chapter 8 of this User Manual.

As noted in Chapter 2.1.3, the illustrative analyses make use of ready-to-use EEG data files, which due to size are not provided in the GitHub repository but are rather downloaded from a separate standalone dataset (B. C. Wang et al., 2025a). Users can run the `illustrative_0_downloadExampleData.m` script to download the files into the `ExampleData` folder if they are not there already. These files are also specified in the repository's `.gitignore` file so that they are not tracked by Git if stored in the user's local instance of the repository. The example data files are described in detail in Chapter 8.1 of this User Manual.

¹⁴Users of previous development versions of MatClassRSA v2 may have encountered a '@' class folder structure rather than '+' package folders. In that case, the user needed to also create an instance of MatClassRSA (e.g., `RSA = MatClassRSA()`) and call functions as e.g., `RSA.Preprocessing.noiseNormalization(X,Y)`. With the current release's package-folder specification, users no longer need to instantiate MatClassRSA and include the instance in function calls.

2.4 Input and Output Specifications

2.4.1 Input Data Specifications

As an analysis toolbox, MatClassRSA does not perform data cleaning; even the data-preparation steps in the Preprocessing module are intended to be applied to already-cleaned data. However, the toolbox *is* designed to work with data in the form that is commonly output by EEGLAB¹⁵ after preprocessing (i.e., 3D space \times time \times trial matrices).

MatClassRSA functions operate on variables that are already loaded to the workspace (rather than filenames). It is therefore the responsibility of the user to have the necessary data already loaded and correctly formatted for input to MatClassRSA functions.

Response Data Matrices, Labels Vectors, Participant Vectors

Functions in the +Preprocessing, +Reliability, and +Classification modules, as well as selected functions in the +RDM_Computation module, operate directly M/EEG input data. These functions generally require at minimum a (1) data matrix and (2) vector of trial labels as inputs. In function documentation, the input matrix of M/EEG response data is usually referred to as *X*, and the input labels matrix is referred to as *Y*. Certain preprocessing functions additionally accept as input a vector of participant identifiers, which we refer to as *P*. These inputs are summarized in Figure 3.

For input data *X*, MatClassRSA functions expect data matrices containing repeated-trials data (generally ranging from tens to thousands of trials per stimulus) from at least two categories (i.e., at least two unique values in the labels vector *Y*). Nearly all functions operating on data will accept a 3D matrix of space-by-time-by-trial data (e.g., as output by EEGLAB) or a 2D matrix of trial-by-feature data in cases where data from single electrodes or single time points are analyzed. If the user intends to analyze data from multiple stimuli and/or participants, the data should already be combined into a single input matrix.

MatClassRSA functions were developed with the primary use case of *X* data matrices containing time-domain evoked responses from multiple electrodes. As a result, the docstrings, manual entries, and illustrative analyses generally reflect this usage and make references to data from different sensors/electrodes and time samples. However, it is also possible to input response data of other formats. For example, data matrices can represent spatial *components* (such as principal components or independent components) rather than sensors along the space dimension, and oscillatory band power or Fourier coefficients as alternative features to time samples. In these cases, users should be mindful to interpret the

¹⁵<https://eeglab.org/>

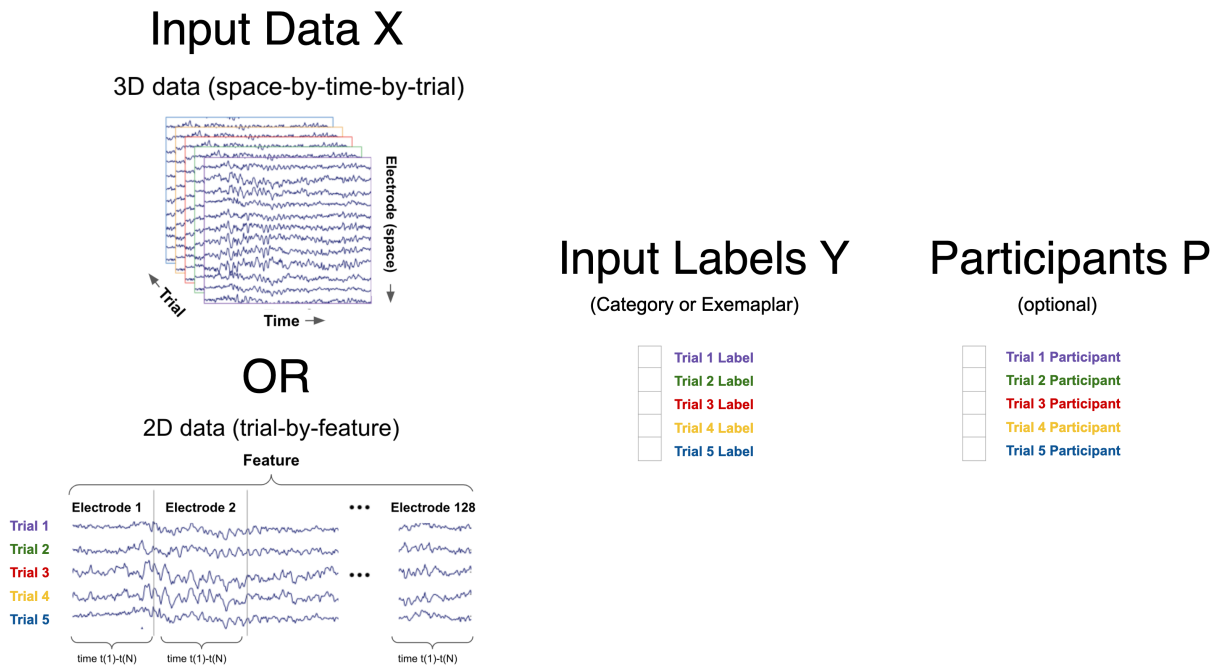


Figure 3: **Primary MatClassRSA data inputs.** Input data matrix X can be shaped as a 3D trial \times feature (typically time) \times trial matrix, or as a 2D trial \times feature matrix. Labels are expected to be input as a numeric vector Y, whose length corresponds to the size of the trial dimension of X. Users may also input a participants vector P for use in selected functions from the Preprocessing module.

structure and features of the data accordingly.

Alongside the input data, most MatClassRSA functions also require the vector of trial labels, generally referred to as Y. This labels vector provides the e.g., stimulus identifier for each trial of response data, and therefore its length should correspond to the size of the trial dimension of X. The ordering of its elements should match the ordering of the data trials of X. MatClassRSA expects Y to be a numeric row or column vector; the values need not start at 1 nor be continuous. When matrices of size $nClasses \times nClasses$ (such as classifier confusion matrices or RDMs) are output by MatClassRSA functions, these matrices' rows and columns will be ordered by the sorted unique elements of the labels vector Y.

The participant identifiers input P is used by Preprocessing functions `shuffleData()` and `averageTrials()` to specify, in cases where the input data matrix contains data from multiple participants, which participant is associated with each trial of data. When input, it is expected to be a vector the same length as Y.

RDMs as Inputs

Users wishing to transform (e.g., symmetrize, rank-order) or visualize proximity matrices may pass such matrices directly into the respective functions. Within the scope of MatClassRSA, proximity matrices are assumed to be square matrices of size $nClasses \times nClasses$. Proximity matrices are often inherently symmetric, such as those containing pairwise classification accuracies or correlation values. However, other proximity matrices—such as multiclass confusion matrices—are not inherently symmetric, and the RDM_Computation module includes functions to eventually symmetrize these matrices for downstream RSA analyses.

2.4.2 Function Outputs

MatClassRSA functions return output variables and/or figures, but do not save any output files. As with data loading, the user can write custom scripts to call MatClassRSA functions and save the outputs into data files.

Module: Preprocessing

3.1 Overview

While MatClassRSA does not include a data-cleaning pipeline, it does include preprocessing functionalities for preparing already-cleaned data for downstream analyses.

The MatClassRSA Preprocessing module includes three functions. First, `shuffleData()` reorders the available trials while keeping each trial paired with its original stimulus label and (if specified) participant identifier. Next, `averageTrials()` computes ‘pseudotrials’, which are averages of trials from the same stimulus and (if specified) participant. Finally, `noiseNormalization()` upweights or downweights the data from each sensor based on its signal-to-noise ratio. These functions can be used in any combination. As noted in Figure 4, Preprocessing functions operate on the input M/EEG data, and function outputs can be input to functions in downstream Reliability, Classification, and RDM Computation modules. Preprocessing functions are also optional in the sense that the input M/EEG data can be directly input to Reliability, Classification, and RDM Computation functions without having undergone any Preprocessing steps.

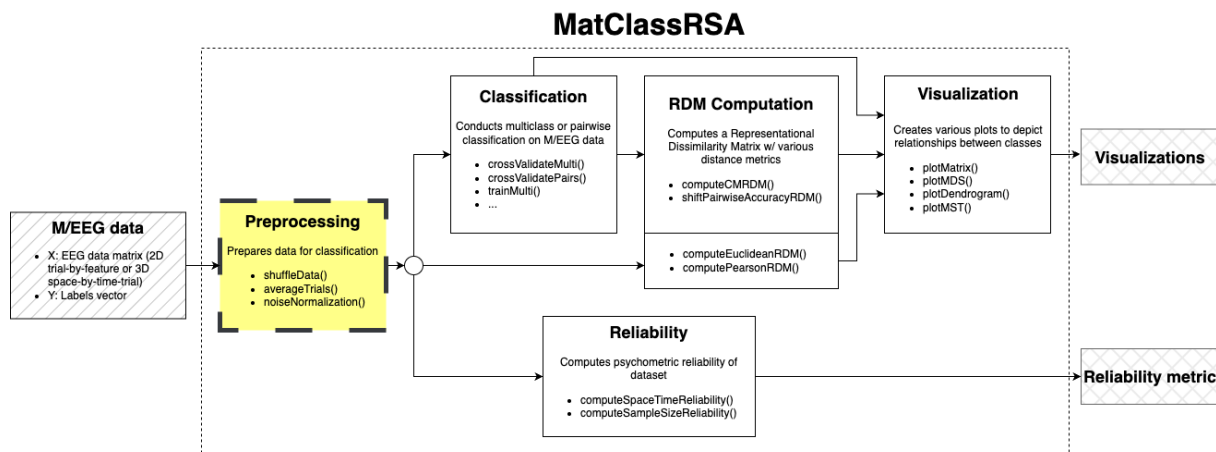


Figure 4: **Position of Preprocessing module in MatClassRSA toolbox.** Preprocessing functions operate on the original input data. Functions from this module can be called prior to calling Reliability, Classification, and RDM Computation functions, though it is not required.

3.2 shuffleData()

This function randomizes, in tandem, ordering of trials in data matrix X, labels in vector Y, and (if specified) participants in vector P. Unlike trial randomization for permutation testing—where the aim is to decouple trials from original stimulus (and participant) labels—for this function global ordering is disrupted, but mappings between trials, stimulus labels, and (if specified) participants are preserved. Therefore, this function is used in cases where users wish to distribute correctly labeled trials across the course of one or more recordings or across participants, prior to trial averaging and downstream analyses.

This function uses the helper function `setUserSpecifiedRng()`.

Syntax

```
[randX, randY, randP, randIdx] = Preprocessing.shuffleData(...
    X, Y, P, rngType);
```

Required inputs

- **X — M/EEG data matrix.** The X matrix contains the response data. The X matrix can be passed into the function in the following shapes:
 - A 3D sensor-by-time-by-trial matrix.
 - A 2D trial-by-feature matrix. For example, this matrix could contain response activations at a single time sample recorded from multiple sensors, or across time from a single sensor. In the function code, 2D data matrices are effectively treated as trial-by-time (single-sensor) matrices.

- **Y — Labels vector.** The Y vector contains the numeric labels corresponding to each trial in the M/EEG data matrix X. Both row and column vectors will be accepted. The length of Y must correspond to the size of X along the trial dimension.

Optional inputs

- **P — Participant vector.** The optional input P specifies the participant identifier of every trial of data. The length of P must correspond to the length of Y and the length of the trial dimension of X. If P is not entered or is empty, the function will return NaN as randomized P. P can be a numeric vector, string array, or cell array.

Optional name-value inputs

- **rngType — Random number generator (rng) specification.** If rngType is not entered or is empty, rng will be assigned here as {'shuffle', 'twister'}. The rngType input can be specified in the following ways:
 - Single acceptable rng specification input (e.g., 1, 'default', 'shuffle'); in these cases, the generator will be set to 'twister'.
 - Dual-argument specifications as either a 2-element cell array (e.g., {'shuffle', 'twister'}) or string array (e.g., ["shuffle", "twister"]).
 - rng struct as previously assigned by rngType = rng.

Outputs

- **randX — Data matrix randomized along the trial dimension.** The output data matrix will be the same size as the input and contains data randomized along the trial dimension according to the ordering provided by randIdx. For a 3D input, this is the third dimension, while for a 2D input, this is the first dimension.
- **randY — Labels vector randomized along the trial dimension.** The output labels vector will be the same size as the input vector and contains trial labels randomized along the trial dimension according to the ordering provided by randIdx.
- **randP — Participant identifier vector randomized along the trial dimension.** If an (optional) participant identifier vector P was entered as an input, the returned variable randP will be the same size as the input vector, containing trial labels randomized along the trial dimension according to the ordering provided by randIdx. If P was not entered or was empty, randP is returned as a single NaN.
- **randIdx — Randomized ordering indices vector.** randIdx contains the ordering that was applied to randomize all of the data inputs. It is a vector the same length as Y.

Example function calls

If the user has no participants vector to input, wishes to use the default rng specification, and does not need the randP output, the function can be called as follows:

```
[randX, randY, ~, randIdx] = Preprocessing.shuffleData(X, Y)
```

In cases where the user has no participants vector to input and does not need the randP output, but does wish to specify rngType, an empty third input can be provided, as follows:

```
[randX, randY, ~, randIdx] = Preprocessing.shuffleData(X, Y, ...  
[], {5, 'philox'})
```

Finally, the user can specify all inputs and obtain all outputs as follows:

```
[randX, randY, randP, randIdx] = Preprocessing.shuffleData(X, Y, ...  
P, {5, 'philox'})
```

A runnable script with example function calls is provided in the ExampleFunctionCalls folder: `example_Preprocessing_shuffleData.m`.

3.3 averageTrials()

This function averages trials in the data matrix X on a per-label basis (as defined by elements of Y) in groups of groupSize trials. In other words, the function averages groups of trials belonging to the same category, where the number of trials averaged in each group is specified by the variable groupSize.

- The user can also optionally enter a vector of participant identifiers P, in which case trial averaging will additionally be performed on a per-participant basis.
- The function takes in optional name-value pairs to specify handling of remainder trials, whether to shuffle the data after averaging (while still retaining the mapping between trials and labels), and to set the random number generator.

If the user wishes to shuffle the ordering of data (while preserving labeling of data observations) prior to averaging, they should call the `shuffleData()` function prior to calling this function.

This function uses the helper functions `cube2trRows()`, `setUserSpecifiedRng()`, and `trRows2cube()`.

Syntax

```
[averagedX, averagedY, averagedP, whichObs] = averageTrials(X, Y, ...  
    groupSize, varargin);
```

Required inputs

- **X — M/EEG data matrix.** The X matrix contains the response data. The X matrix can be passed into the function in the following shapes:
 - A 3D sensor-by-time-by-trial matrix.
 - A 2D trial-by-feature matrix. For example, this matrix could contain response activations at a single time sample recorded from multiple sensors, or across time from a single sensor. In the function code, 2D data matrices are effectively treated as trial-by-time (single-sensor) matrices.
- **Y — Labels vector.** The Y vector contains the numeric labels corresponding to each trial in the M/EEG data matrix X. Both row and column vectors will be accepted. The length of Y must correspond to the size of X along the trial dimension.
- **groupSize — Number of single trials to be averaged in each group.** For example, a groupSize value of 5 would mean that groups of 5 trials are averaged, *not* that the available trials are partitioned into 5 groups total.

Optional inputs

- **P — Participant vector.** The optional input P specifies the participant identifier of every trial of data. The length of P must correspond to the length of Y and the length of the trial dimension of X. If P is not entered or is empty, the function will return NaN as randomized P. P can be a numeric vector, string array, or cell array.

Optional name-value inputs

- **rngType — Random number generator (rng) specification.** If rngType is not entered or is empty, rng will be assigned here as {'shuffle', 'twister'}. The rngType input can be specified in the following ways:
 - Single acceptable rng specification input (e.g., 1, 'default', 'shuffle'); in these cases, the generator will be set to 'twister'.
 - Dual-argument specifications as either a 2-element cell array (e.g., {'shuffle', 'twister'}) or string array (e.g., ["shuffle", "twister"]).
 - rng struct as previously assigned by rngType = rng.
- **handleRemainder — method to handle remainder trials.** For example, if you have 21 rows with label 1 and set average group size to 5, you would have 4 groups (20/5), and 1 remainder row with label 1. If not specified, defaults to 'discard'. Options:

- 'discard' — Disregard the remaining data (default).
 - 'newGroup' — Average all remaining trials in a new row, despite these trials not fulfilling the group size.
 - 'append' — Append the remaining data to the last averaged row of the same label.
 - 'distribute' — Distribute the remaining data to different groups of the same label.
- **endShuffle** — **whether to shuffle the data after averaging.** This shuffling process preserves the mapping of data to corresponding labels and participants. This step is recommended as the main function loops through participants (if input) and stimulus labels during computation of averages, which groups the output averages by participants (if input) and stimulus labels. Therefore, this option redistributes observations from each stimulus category (e.g., as input to cross-validated classification) across the set of observations. If not specified, defaults to 1. Options:
 - 1: Perform end shuffling (default).
 - 0: Do not perform end shuffling.

Outputs

- **averagedX** — **Data matrix after trial averaging.** The output data matrix will be the same shape (2D or 3D) as the input data matrix but will have a smaller trial dimension than the input data matrix. averagedX contains pseudotrials averaged in sets of groupSize (and possible extra pseudotrial(s) for remainder trials, if specified).
- **averagedY** — **Labels vector for output pseudotrials.** The length of averagedY will correspond to the size of the trial dimension of the output data matrix.
- **averagedP** — **Participant identifiers for output pseudotrials.** averagedP will be a vector the same length as averagedY. Note: If P was not entered or was empty, trial averaging would not have considered participant identifiers, and all values of averagedP will be zero.
- **whichObs** — **Trial averaging information matrix.** This matrix is of size length(averagedY) by groupSize. The matrix elements denote which trials from the input matrix X were averaged in each output pseudotrial of averagedX.

Example function calls

If the user would like to average trials in groups of 5; has no participants vector to input and does not need the averagedP output; and wishes to use all default specifications of the name-value inputs, the function can be called as follows:

```
[averagedX, averagedY, ~, whichObs] = averageTrials(X, Y, 5)
```

If the user would like to specify the participants vector and obtain that output, while using all default specifications for name-value inputs, the function call would look like this:

```
[averagedX, averagedY, averagedP, whichObs] = averageTrials(X, Y, ...
5, P)
```

If the user would like to customize specific name-value inputs, only those inputs being specified need to be input to the function. For example, to override the default rngtype, the user could call the function as follows:

```
[averagedX, averagedY, ~, whichObs] = averageTrials(X, Y, ...
5, 'rngtype', {5, 'philox'})
```

Multiple name-value inputs can also be specified, in any order, after optional inputs (if used):

```
[averagedX, averagedY, P, whichObs] = averageTrials(X, Y, ...
5, P, 'endShuffle', 0, 'handleRemainder', 'append')
```

A runnable script with example function calls is provided in the ExampleFunctionCalls folder: `example_Preprocessing_averageTrials.m`.

3.4 noiseNormalization()

When recording brain responses using EEG or MEG, data from individual sensors typically vary in terms of signal-to-noise ratio (SNR). The `noiseNormalization()` function effectively ‘downweights’ sensors that have low SNR and ‘upweights’ sensors that have high SNR. This preprocessing step also uses information regarding how the electrodes may vary with each other. This is quantified by computing, for each timepoint, the covariance matrix across electrodes for a single condition on a data matrix of dimensions $nTrials \times nElectrodes$ (and averaged across time).

This function is adapted from code provided by Guggenmos et al. (2018).¹⁶ This function uses the helper function `cov1para()`.

Important note for MatClassRSA users: Guggenmos et al. (2018) showed that preprocessing with multivariate noise normalization improves the reliability of representational dissimilarity matrices for many similarity measures. However, while the function helps

¹⁶https://github.com/m-guggenmos/megmvpa/blob/master/tutorial_matlab/matlab_distance.ipynb

to de-emphasize local covariances that may result from electrical noise, it appears that in some cases, usage of this function alongside MatClassRSA classification functions also has the potential to negatively impact classification accuracy (perhaps through perturbing biologically relevant covariance). Therefore, users are cautioned to assess impacts of this function in downstream classification analyses. For more information, see Illustrative Analysis 1 (Chapter 8.2).

Syntax

```
[normData, sigmaInv] = Preprocessing.noiseNormalization(X, Y);
```

Required inputs

- **X — M/EEG data matrix.** The X matrix contains the response data. The X matrix can be passed into the function in the following shapes:
 - A 3D sensor-by-time-by-trial matrix.
 - A 2D trial-by-feature matrix. For example, this matrix could contain response activations at a single time sample recorded from multiple sensors, or across time from a single sensor. In the function code, 2D data matrices are effectively treated as trial-by-time (single-sensor) matrices.
- **Y — Labels vector.** The Y vector contains the numeric labels corresponding to each trial in the M/EEG data matrix X. Both row and column vectors will be accepted. The length of Y must correspond to the size of X along the trial dimension.

Outputs

- **normData — Noise-normalized data matrix.** It will be the same size as the input data matrix.
- **sigmaInv — Inverse of the square root of covariance matrix.** This is a matrix of size $nElectrodes \times nElectrodes$ used to normalize the data.

Example function call

This function call will involve only the two required inputs:

```
[normData, sigmaInv] = Preprocessing.noiseNormalization(X, Y);
```

A runnable script with example function calls is provided in the ExampleFunctionCalls folder: `example_Preprocessing_noiseNormalization.m`.

Module: Reliability (Data Quality Assessments)

4.1 Overview

Users may wish to compute the reliability of the data, as data reliability affects further downstream data analyses and also places an upper bound on how well any model can explain the data. Specifically, reliability is a metric that quantifies the similarity of a measurement across multiple repeats of an input. For example, computing an electrode's reliability will quantify its response similarity across multiple presentations of a single stimulus. This metric is similar to the signal-to-noise ratio (SNR).

To compute reliability of electrodes on a dataset where there are multiple trials (i.e., repetitions) for each stimulus (i.e., a dataset of dimensions $N_{\text{stimuli}} \times N_{\text{trials}}$ for each electrode), we first split the N_{trials} in half, resulting in two data partitions, each with half the total number of trials. Each of the two partitions is then averaged across the trials dimension, resulting in two vectors of dimensions N_{stimuli} . The reliability, R , is then computed as the correlation between those two vectors. Since only half of the total trials are used in each partition, this estimate of reliability would be lower than if *all* the trials were used (which is typical in any analysis that uses data that are averaged across all trials). To correct for this, we apply the Spearman-Brown correction, where $R_{\text{corrected}} = \frac{2R}{1+R}$. Since these functions ultimately involve correlations over vectors of N_{stimuli} , they are recommended for datasets with larger stimulus sets (in at least the tens or hundreds).

This module of MatClassRSA includes two functions to compute the reliability of the data. The first function, `computeSpaceTimeReliability()` computes reliability using all available data at each response feature, where a response feature is typically a sensor and/or time sample. Next, `computeSampleSizeReliability()` computes reliability for a single sensor or time point over a varying trial sample size. As illustrated in Figure 5, functions in this module can operate on the original input data, or can operate on data output by one or more functions from the Preprocessing module. The Reliability functions do not directly feed into any downstream MatClassRSA modules, but the reliability measures may be useful for interpreting Classification results or comparisons between RDMs.

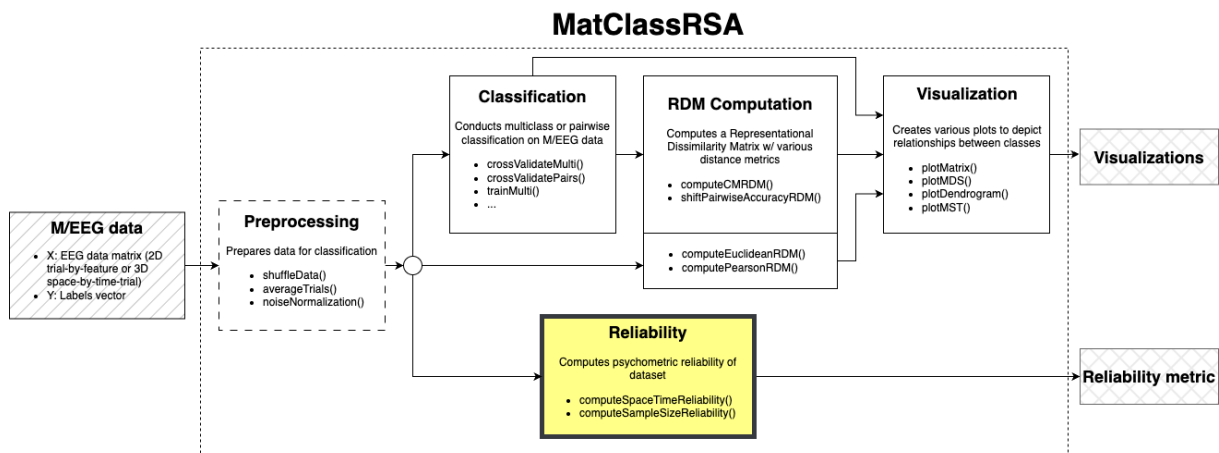


Figure 5: **Position of Reliability module in MatClassRSA toolbox.** Reliability functions can be called directly on the input data, or on data that has undergone one or more steps from the Preprocessing module. Reliability outputs do not intersect with functions from the remaining modules.

4.2 computeSpaceTimeReliability()

This function returns split-half reliabilities computed for each sensor (component) across time. This computation is performed across a specified number of permutations, where each permutation is associated with a random shuffling of the data across trials (i.e., the trial indices are randomized in each permutation). This randomization procedure allows for different split-halves to be constructed for each permutation. Each split-half is then averaged across trials and correlated with each other to obtain a reliability estimate for that permutation.

With the resulting data matrix of split-half reliabilities, one can take the mean along the third (sensor/component) dimension and this will tell you the average reliability across

components at each time point. On the other hand, if one takes the mean across the first dimension (the time axis), one will be able to see how reliable each sensor/component is across time (on average). Since split-half reliability is computed on *half* the total number of trials, the estimate of the dataset's reliability is lower than that computed using *all* the trials. Spearman-Brown correction accounts for this and estimates reliability if all the trials could be used.

This function uses the helper functions `computeReliability()` and `setUserSpecifiedRng()`.

Syntax

```
[reliabilities] = Reliability.computeSpaceTimeReliability(X, Y, ...  
    varargin);
```

Required inputs

The function has two required inputs: Response data matrix X and labels vector Y.

- **X — M/EEG data matrix.** The X matrix contains the response data. The X matrix can be passed into the function in the following shapes:
 - A 3D sensor-by-time-by-trial matrix.
 - A 2D trial-by-feature matrix. For example, this matrix could contain response activations at a single time sample recorded from multiple sensors, or across time from a single sensor. In the function code, 2D data matrices are effectively treated as trial-by-time (single-sensor) matrices.
- **Y — Labels vector.** The Y vector contains the numeric labels corresponding to each trial in the M/EEG data matrix X. Both row and column vectors will be accepted. The length of Y must correspond to the size of X along the trial dimension.

Optional name-value inputs

- **numPermutations — Number of split-half reliability permutations.** This input specifies the number of times to randomly partition the trials and should be an integer greater than zero. If numPermutations is not entered or is empty, this defaults to 10.
- **rngType — Random number generator (rng) specification.** If rngType is not entered or is empty, rng will be assigned here as {'shuffle', 'twister'}. The rngType input can be specified in the following ways:
 - Single acceptable rng specification input (e.g., 1, 'default', 'shuffle'); in these cases, the generator will be set to 'twister'.

- Dual-argument specifications as either a 2-element cell array (e.g., {'shuffle', 'twister'}) or string array (e.g., ["shuffle", "twister"]).
- rng struct as previously assigned by `rngType = rng`.

Outputs

- **reliabilities** — **Matrix of sensor reliabilities across time.**

For 3D input matrices, the dimensions of `reliabilities` will be space-by-time-by-permutation. One would typically average across the permutations and sensors dimensions in order to obtain a time course of average reliability across sensors. For 2D input matrices, the dimensions of `reliabilities` will be time-by-permutation. In this case, the data are assumed to contain only one sensor (component) and would average across the permutations dimension to obtain a time course of average reliability.

Example function calls

The following function call will use default specifications for `numPermutations` and `rngType`:

```
reliabilities = Reliability.computeSpaceTimeReliability(X, Y);
```

If the user wishes to specify `numPermutations` — with 100 permutations, for example — but use the default specification for `rngType`, the function can be called as follows:

```
reliabilities = Reliability.computeSpaceTimeReliability(X, Y, ...  
'numPermutations', 100);
```

If the user wishes to use a custom specification for `rngType` — for example, setting it to {5, 'philox'} — while using the default value of `numPermutations`, the function can be called as follows:

```
reliabilities = Reliability.computeSpaceTimeReliability(X, Y, ...  
'rngType', {5, 'philox'});
```

Finally, the user can specify all four inputs as follows:

```
reliabilites = Reliability.computeSpaceTimeReliability(X, Y, ...  
'numPermutations', 100, 'rngType', {5, 'philox'});
```

A runnable script with example function calls is provided in the `ExampleFunctionCalls` folder: `example_Reliability_computeSpaceTimeReliability.m`.

4.3 computeSampleSizeReliability()

Reliability is known to vary according to the size (i.e., number of trials per stimulus) of the data. This function computes split-half reliability across varying trial subset sizes. Typically, one would aggregate the trials across participants and provide the aggregated data as input into this function. A typical use case would be to average, for each trial subset size, the output reliabilities across the sensor (component) dimension at a fixed time sample or across the time dimension at a fixed sensor, as well as across all permutations. This will produce a vector whose entries indicate the average reliability across sensors or time, respectively, as a function of trial subset size. Since split-half reliability is computed, the Spearman-Brown correction is applied to estimate the reliability across the entire data at once.

This function uses the helper functions `computeReliability()` and `setUserSpecifiedRng()`.

Syntax

```
reliabilities = Reliability.computeSampleSizeReliability(X, Y, ...  
    featureIdx, varargin);
```

Required inputs The function has two required inputs: Response data matrix X and labels vector Y.

- **X — M/EEG data matrix.** The X matrix contains the response data. The X matrix can be passed into the function in the following shapes:
 - A 3D sensor-by-time-by-trial matrix.
 - A 2D trial-by-feature matrix. For example, this matrix could contain response activations at a single time sample recorded from multiple sensors, or across time from a single sensor. In the function code, 2D data matrices are effectively treated as trial-by-time (single-sensor) matrices.
- **Y — Labels vector.** The Y vector contains the numeric labels corresponding to each trial in the M/EEG data matrix X. Both row and column vectors will be accepted. The length of Y must correspond to the size of X along the trial dimension.
- **featureIdx — Index of data feature at which reliability for each sensor (component) is desired.** The featureIdx argument is an integer that indicates the time sample at which to compute reliability across sensors, or the sensor at which to compute reliability across time samples, while reliability is also computed across different sizes of trial subsets.
 - featureIdx should be between 1 and the feature length in the data (i.e., number of time samples or number of sensors), inclusive.

- If a 3D matrix is input, `featureIdx` specifies the index at which the data are subset along the *second* dimension. Thus, if the user inputs a space-by-time-by-trial matrix, `featureIdx` specifies the time point at which reliabilities will be computed and averaged across all sensors. If the user wishes to specify the sensor rather than the time point, the dimensions of the input data matrix should be permuted so that space is represented along the second dimension (see example function call below). If a 2D trial-by-feature matrix is input, this input specifies the index at which the data are subset along the second (feature) dimension.

Optional name-value inputs

- **numTrialsPerHalf — Number of trials used in split-half reliability calculations.** This vector specifies how many trials to include in a split half for each reliability computation. For example, `[1,2,3]` would correspond to using 2, 4, and 6 trials in the reliability computations. If `numTrialsPerHalf` is not entered or is empty, this defaults to `[1]`, meaning that 2 trials total are used.
- **numPermutations — Number of split-half reliability permutations.** This input specifies the number of times to randomly partition the trials and should be an integer greater than zero. This is for the inner loop to compute reliability. If `numPermutations` is not entered or is empty, this defaults to 10.
- **numTrialPermutations — Number of data-selection rounds for reliability calculations.** This input specifies how many times to choose trials in the data set to compute reliability. This is for the outer loop. This is useful if we want to compute the variance of the reliability across random draws of the trials. If `numTrialPermutations` is not entered or is empty, this defaults to 10.
- **rngType — Random number generator (rng) specification.** If `rngType` is not entered or is empty, `rng` will be assigned here as `{'shuffle', 'twister'}`. The `rngType` input can be specified in the following ways:
 - Single acceptable `rng` specification input (e.g., `1`, `'default'`, `'shuffle'`); in these cases, the generator will be set to `'twister'`.
 - Dual-argument specifications as either a 2-element cell array (e.g., `{'shuffle', 'twister'}`) or string array (e.g., `["shuffle", "twister"]`).
 - `rng` struct as previously assigned by `rngType = rng`.

Outputs

- **reliabilities — Matrix of reliabilities across number of trials.** If input matrix was 3D, the output dimensions `numTrialPermutations x length(numTrialsPerHalf) x`

nSpace. If input matrix was 2D, the output dimensions are numTrialPermutations x length(numTrialsPerHalf). Note that the permutations used to split the trials in half for the inner loop reliability computation have already been averaged.

Example function calls

Call the function with 2D or 3D input data matrix, feature/time point 5, and default specifications for optional inputs:

```
reliabilities = Reliability.computeSampleSizeReliability(X, Y, 5)
```

To compute reliabilities at a single sensor (rather than time point) when calling the function, permute the first two dimensions of the input data matrix so that its dimensions are time-by-space-by-trial. The following function thus computes reliabilities at sensor 96:

```
reliabilities = Reliability.computeSampleSizeReliability(...  
permute(X, [2, 1, 3]), Y, 96);
```

Call the function with 2D or 3D input data matrix, feature/time point 5, custom rng specification, and otherwise default inputs:

```
reliabilities = Reliability.computeSampleSizeReliability(X, Y, 5, ...  
'rngType', {rngSeed, 'philox'});
```

Call the function with 2D or 3D input data matrix, feature/time point 5, an increasing number of trials to use, from 1 to 10 in each split-half partition, and otherwise default inputs:

```
reliabilities = Reliability.computeSampleSizeReliability(X, Y, 5, ...  
'numTrialsPerHalf', 1:10);
```

Call the function with 2D or 3D input data matrix, feature/time point 5, an increasing number of trials to use, from 1 to 10 in each split-half partition, 20 split-half permutations, 25 data-selection permutations, and default rng:

```
reliabilities = Reliability.computeSampleSizeReliability(X, Y, 5, ...  
'numTrialsPerHalf', 1:10, 'numPermutations', 20, ...  
'numTrialPermutations', 25);
```

A runnable script with example function calls is provided in the ExampleFunctionCalls folder: example_Reliability_computeSampleSizeReliability.m.

Module: Classification

5.1 Overview

The MatClassRSA Classification module provides functionalities for various machine learning classifiers to predict categorical labels from M/EEG data. As noted in Figure 6, functions in this module operate on M/EEG data, which could optionally have first been passed through one or more functions from the Preprocessing module. Classification module outputs can be passed into RDM Computation and Visualization functions.

Classification is supervised learning problem, where the goal is to predict the category of an observation of data. To accomplish this, a statistical model is trained on data observations that are each accompanied by category labels, and then the model is used to classify new, unseen observations into the existing categories. In the context of MatClassRSA, the training data typically represents single or group-averaged trials of M/EEG data, whereas the labels refer to some specification of the stimuli—which, broadly defined, could include e.g., stimulus descriptors, task labels, or descriptors of the participant.

MatClassRSA provides options for running iterative cross-validation analyses on data (to evaluate the performance of a classification model) or training a model once to perform classifications on external data. There are two options in terms of partitioning data for classification: Multiclass and pairwise. For the multiclass options, a single classification

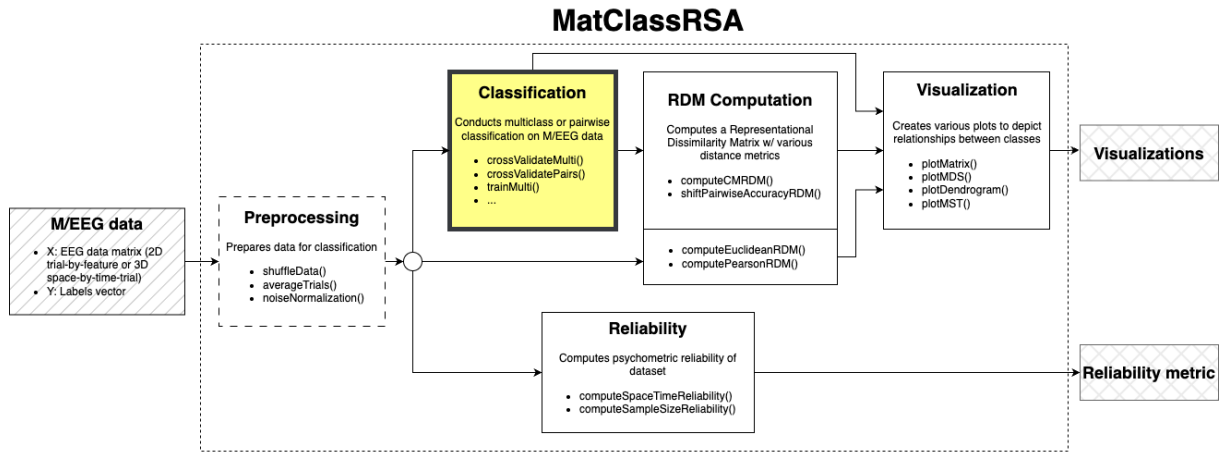


Figure 6: **Position of Classification module in MatClassRSA toolbox.** Classification functions can operate on the original input directly, or on the data that have undergone Preprocessing steps. Classification outputs can be input to RDM Computation and Visualization modules.

model is trained to discriminate between all N unique classes provided in the dataset. For the pairwise option, a separate model is trained for every pair of unique classes, where there are $\binom{N}{2}$ pairs for N unique classes. Finally, MatClassRSA provides the option to optimize the model prior to classification (see functions with the ‘_opt’ subscript). Optimization is currently only available for the SVM classifier, and is conducted via grid search.

Thus, the available classification functions are as follows:

Multiclass classification

- `crossValidateMulti()`
- `crossValidateMulti_opt()`
- `trainMulti()/predict()`
- `trainMulti_opt()/predict()`

Pairwise classification

- `crossValidatePairs()`
- `crossValidatePairs_opt()`
- `trainPairs()/predict()`
- `trainPairs_opt()/predict()`

These functions can be found in the MatClassRSA `src/+Classification` directory. In this chapter, in addition to the standard function descriptions and documentation, we also include flowcharts summarizing the workflow of each user-called function, due to their increased complexity.

5.2 High-level Overview: M/EEG Classification for RSA

In an RSA context, analyses are performed over dissimilarity matrices (RDMs) which may reflect different data modalities such as various forms of neuroimaging data, computational

models, and perceptual measures. Classification offers several advantages as a means of computing RDMs from M/EEG data:

- **Multivariate:** M/EEG is often a high-dimensional form of data, especially when data from multiple sensors and time points are available. Performing mass-univariate analyses (e.g., at every sensor and/or time point) can deplete the statistical significance of the results after multiple comparison corrections are applied. Classification is one approach toward circumventing this issue, as a single classification analysis can incorporate data from multiple electrodes and features (e.g., time samples, Fourier coefficients) at once, and capture distributed category-related activity. Classification is thus one way of utilizing the full neural response at once to obtain measures of similarity or difference between stimulus categories.
- **Data driven:** While many EEG analyses rely on preselected electrodes, electrode groups, or time intervals of interest for analyses, classification is data-driven in that the algorithm will selectively weight the features of the input data according to their utility in maximizing decodability among the categories. In addition, classification can be used in a ‘searchlight’ configuration (Su et al., 2012), to identify spatial and temporal/frequency-related components whose activations differ most between stimulus categories.
- **Predictive:** Predicting labels of unseen observations is generally a more challenging task than descriptive analyses (such as reporting averages or other summary statistics), because the analysis approach must now generalize to new observations. However, predictive approaches may better convey the generalizability of a finding, and is crucial in settings where correctly labeling new data is important (e.g., for clinical assessment and diagnosis).
- **Extending experimental paradigms:** Classification is one approach toward facilitating studies involving large stimulus sets (10s-100s of stimuli). In the RSA context, too, classification is one step toward directly comparing M/EEG data with other other representations of the stimulus set (e.g., perceptual responses, computational models of stimuli) in similarity space.

Classification is by no means the only way to compute M/EEG RDMs. See Chapter 6 for options to compute RDMs directly from the input M/EEG data as opposed to classification outputs.

5.3 Overview of MatClassRSA Classification Specifications

5.3.1 Core inputs: Data Matrix and Labels Vector

Two primary inputs are required by MatClassRSA: stimuli response data X and its corresponding labels vector Y . X can be either 2D (trial-by-feature) or 3D (space-by-time-by-trial) data, while Y is a single column vector containing labels for each trial of the X matrix.¹⁷

5.3.2 Subsetting Data Features

Users can choose to select a specific subset of data over which to run their analysis. If X is passed in as a 3D space-by-time-by-trial matrix, then the user can use subsetting to run their analyses over a specific spatial and/or temporal region of the response, as opposed to the entire dataset. If X is passed in as a 2D trial-by-feature matrix, MatClassRSA allows subsetting along the feature dimension. Subsetting will have the added benefit of reducing analysis time.

MatClassRSA classification functions contain the optional input arguments `spaceUse`, `timeUse`, and `featureUse`. If X is passed in as a 3D space-by-time-by-trial matrix, then the `spaceUse` and `timeUse` arguments can be used to select specific space and time indices over which to conduct the analyses. If X is passed in as a 2D trial-by-feature matrix, then `featureUse` can be used to select the feature indices over which to conduct the analyses.

5.3.3 Classifier Selection

MatClassRSA provides the following classifiers:

- Support Vector Machine (SVM)
- Random Forest (RF)
- Linear Discriminant Analysis (LDA)

For the SVM classifier, two kernels are supported by MatClassRSA: The linear kernel and radial basis function (RBF) kernel. As for random forest, the two hyperparameters supported are `numTrees` (controls the number of decision trees to grow) and `minLeafSize` (controls the minimum number of observations per leaf). The MatClassRSA implementation of LDA does not have any tunable hyperparameters. For more information on the SVM classifier, see Hsu, Chang and Lin's 2003 paper, "A Practical Guide to Support Vector Classification". For more info on hyperparameters for random forest, see Matlab documentation for the

¹⁷In practice, any X passed in as a 3D matrix is ultimately reshaped by MatClassRSA into a 2D matrix as input to the core classification functions.

`treeBagger()` class.¹⁸ For more information on the LDA classifier, please see the Matlab documentation for the `fitcdiscr()` function.¹⁹

Note for SVM classifications: If, when performing any SVM classification, 1) you receive an error message regarding an SVM function not being found, 2) the function hangs (no changes to the pop-up progress bar for several minutes), or 3) classifier performance is unexpectedly poor, you may still need to set up LIBSVM. See Chapter 2.2.3 for more information.

5.3.4 Classifier Optimization

Currently, MatClassRSA supports hyperparameter optimization only with the SVM classifier. Optimization is conducted via grid search over the variables C (for both linear and rbf kernels) and γ (gamma; for the rbf kernel only). A grid search is conducted by choosing a vector of values for each hyperparameter to search over, then evaluating each combination of hyperparameters to find the one that produces the highest cross validation accuracy. Thus, grid search can be thought of as a brute-force approach for hyperparameter optimization, and users should be aware that it may be time-consuming.

The default grid used for search is 5 logarithmically spaced points between 10^{-5} and 10^5 . To run classification with optimization, users can select the classification functions containing the ‘_opt’ subscript. Note that when using the train/predict functions, only the train functions should contain the ‘_opt’ subscript.

5.3.5 Principal Components Analysis (PCA)

Principal Components Analysis (PCA) is a linear transformation that can be applied to brain-imaging data that reduces data dimensionality and extracts the principal components, which are linear combinations of the original features. Principal Components are orthogonal and are returned in descending order of variance explained. In the context of MatClassRSA, PCA can speed up processing by reducing the size of the data while preserving the most significant information that allows for robust classification performance. Analyzing the principal components can also inform us about the spatial/temporal features corresponding to the most relevant response to the stimuli.

In MatClassRSA, PCA is applied to input observation matrix X . If X is a 3D space-by-time-by-trial matrix, then PCA is applied after reshaping X into a 2D trial-by-feature matrix. The PCA input argument has a default value of 0.99, which means that the selected principal components are chosen to explain 99% of the variance of the data. As PCA is an unsupervised learning technique (in that the stimulus labels are not involved in the calculation),

¹⁸<https://www.mathworks.com/help/stats/treebagger.html>

¹⁹<https://www.mathworks.com/help/stats/fitcdiscr.html>

it is acceptable to compute it over the complete input data prior to partitioning and cross-validation train-test iterations. MatClassRSA computes PCA on the complete data by default, but users can also specify PCA to be computed separately over each training partition during cross validation.

5.3.6 Data Centering and Scaling

Machine-learning techniques are often sensitive to the magnitude of input data. For example, if data features from disparate sources (e.g., M/EEG, EKG, and behavioral responses) were combined in a classification task, classifiers could exhibit a bias toward the data features with larger values.

MatClassRSA provides two standard steps to mitigate this issue: Centering and scaling. Centering is computed by subtracting the mean from each feature, which forces all features to have 0 mean. Scaling is done by applying a scalar such each feature has a standard deviation of 1.

The classification functions provided by MatClassRSA accept the optional input arguments `center` and `scale`, which accept `true` or `false` arguments to enable or disable centering and scaling. The default value for centering is `true`. However, the default value for scaling is `false`. This is because multi-sensor data from a given participant are already assumed to be on the same data scale, and hence important information might be lost by scaling the data. However, users may wish to set scaling to `true` if, for example, combining multimodal data in the input matrix, classifying data from multiple participants, or training and testing classifier models on different participants.²⁰

5.3.7 Cross Validation

Cross validation is a standard technique in machine learning to evaluate how a model generalizes to unseen data. To accomplish this, the data are split into N partitions (also known as folds), where $N - 1$ folds act as training data to build the model, and the remaining fold is used as a test set to evaluate classification performance. This process is repeated until every fold has been used as testing data.

MatClassRSA supports cross-validation analysis via the classification functions containing the text 'crossValidate' (`crossValidateMulti()`, `crossValidatePairs()`, etc.). These functions also offer configurable parameters such as the number of cross-validation folds.

²⁰Noise normalization may also help with this; see Chapter 8.2 for more information.

5.3.8 Permutation testing

When performing classification analyses, it is often helpful to be able to estimate how likely an observed result is under the null hypothesis. In the case of classification, the null hypothesis is that no meaningful relationship exists between the data observations and their associated labels; hence, in an N -class classification context where equal numbers of observations were analyzed for each stimulus category, chance-level classification accuracy would be $1/N$. However, it can be difficult to determine whether a classification result is sufficiently ‘good’ or high based on accuracy alone.

Permutation testing is a procedure for empirically generating null distributions against which observed results can be compared. MatClassRSA’s permutation testing is performed in alignment with the aforementioned null hypothesis, whereby the null distribution for a given classification task is created empirically by performing many classifications of the input data, but with the ordering of labels in the input labels vector Y randomized independently of the order of the observations in the input data matrix X . This procedure provides a distribution of classifier outputs for data instances that truly lack any meaningful connection to their labels. One fundamental advantage of permutation testing is that it is a nonparametric test, meaning that no underlying distribution is assumed for the data; permutation testing is also a flexible approach to assessing statistical significance, in that it can be set up as needed for the data and hypothesis at hand (Golland and Fischl, 2003).

MatClassRSA offers simple (per-run) statistical significance assessment using permutation testing. We refer to each classification of disrupted-label data as a *permutation* or *permutation iteration*. It is helpful to perform enough permutation iterations to obtain a smooth null distribution—e.g., at least 100 or up to 10,000 iterations (Golland and Fischl, 2003). The original classification accuracy (with the correctly labeled data) is then compared to this distribution to empirically determine the likelihood of getting the result by chance, which in this case is the percentage of null-distribution observations exceeding the observed value. This likelihood is the p-value of the classification. A p-value of 0.05 is often chosen as a threshold for a statistically significant result. At this time, MatClassRSA does not offer functionalities for multiple comparison correction.

Statistical significance for M/EEG classifications has also been computed using parametric tests, such as the Binomial distribution (e.g., Kaneshiro et al. (2015b)); this method is additionally much faster than performing the many classification iterations required for permutation testing. However, it has been noted that parametric estimation of p-values can be biased due to data reuse across cross-validation iterations (Noirhomme et al., 2014). For this reason, the v2 release of MatClassRSA no longer offers p-value calculation via the Binomial distribution approach.

In MatClassRSA, permutation testing is conducted by passing the optional name-value input argument `permutations` into any of the following functions:

- `crossValidateMulti()`
- `crossValidateMulti_opt()`
- `crossValidatePairs()`
- `crossValidatePairs_opt()`
- `predict()`

Users can set the `permutations` variable as a positive integer to determine the number of permutations to run. In the case of multiclass classification, the output struct contains the field `pVal`, which contains a single p-value from permutation testing, and the field `permAccs`, which is the distribution of accuracies computed from shuffling the data. For pairwise classification, the output struct will contain a field `pValMat`, which is a matrix containing the p-value corresponding to the decision boundary for each pair of classes. It will also contain `permAccMat`, which is a 3D matrix of permutation testing accuracies for each decision boundary (dimensions are `label1`, `label2`, `permutation`). From these outputs the user can assess statistical significance based on their p-value threshold of choice.

Note about MatClassRSA implementations: Users are advised to consult the function specifications in this chapter, as permutation testing implementations vary by function. Permutation testing in this toolbox is carried out overall by permuting labels of *only* the training data, while labels of test data remain intact. This configuration addresses the question of how well a classifier trained on randomly labeled training observations is able to predict true labels of test observations. Specific implementations vary by function, and users are advised to consult the documentation for each function for complete descriptions. Generally, for speed, accuracies of each permutation iteration may be based on a single train-test partition only. If users seek different permutation testing configurations than are offered in the functions, they can conduct permutation testing manually as follows for each permutation iteration:

1. Generate an instance of surrogate data by e.g., randomizing elements in the labels vector only;
2. Input the surrogate data to a function, with permutation testing turned off.

The user can then save the outputs over a desired number of function calls (permutation iterations) to serve as the null distribution.

5.3.9 Classifier outputs

For the cross validation functions (`crossValidateMulti()`, `crossValidateMulti_opt()`, `crossValidatePairs()`, `crossValidatePairs_opt()`), output is formatted into the struct `C`, which contains the results of the cross validation, permutation testing, and other related information.

For the multiclass cross validation functions (`crossValidateMulti()` and `crossValidateMulti_opt()`), `C` contains a confusion matrix, which describes the actual labels of each observation compared to its predicted class, along with the predicted labels for each class and total classification accuracy.

As for the pairwise cross validation functions (`crossValidatePairs()`, `crossValidatePairs_opt()`), `C` will contain the $\binom{n}{2}$ length cell matrix `pairwiseInfo`, where each elements contains classification related info for the decision boundary corresponding to a certain pair of labels. `C` will also contain the accuracy matrix `AM`, where each off-diagonal element represents the accuracy of a decision boundary for classifying two classes.

For the model output functions (`trainMulti()`, `trainMulti_opt()`, `trainPairs()`, `trainPairs_opt()`), the output `M` is simply the classification model obtained by training on the input data, which can then be passed in the `predict()` function along with new observations to predict the labels of those new observations.

The output of `predict()` is the struct `P`, which has similar outputs as the multiclass cross validation functions.

Note that functions with the ‘_opt’ subscript will also return optimized hyperparameter values.

If permutation testing is specified, `C` and `P` will contain the the p-value(s) calculated during permutation testing and a distribution of permutation testing accuracies. Please see the following section for more details on permutation testing output.

5.4 crossValidateMulti()

Given a data matrix `X` and labels vector `Y`, this function will conduct multiclass cross validation, then output a struct containing the classification accuracy, confusion matrix, and other related information. Optional name-value parameters can be passed in to specify

classification related options.

This function uses the helper functions (which can be found in the +Utils folder): `initInputParser()`, `checkInputDataShape()`, `verifySVMParameters()`, `subsetTrainTestMatrices()`, `setUserSpecifiedRng()`, `processTrainDevTestSplit()`, `trainDevTestPart()`, `cvData()`, `fitModel()`.

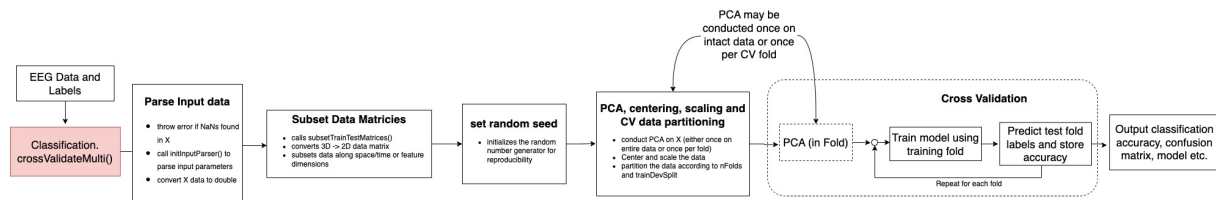


Figure 7: Overview of `crossValidateMulti()` functionality.

Syntax

```
C = crossValidateMulti(X, Y, varargin);
```

Required inputs

- **X — M/EEG data matrix.** The X matrix contains the response data. The X matrix can be passed into the function in the following shapes:
 - A 3D sensor-by-time-by-trial matrix.
 - A 2D trial-by-feature matrix. For example, this matrix could contain response activations at a single time sample recorded from multiple sensors, or across time from a single sensor. In the function code, 2D data matrices are effectively treated as trial-by-time (single-sensor) matrices.
- **Y — Labels vector.** The Y vector contains the numeric labels corresponding to each trial in the M/EEG data matrix X. Both row and column vectors will be accepted. The length of Y must correspond to the size of X along the trial dimension.

Optional name-value inputs

- **timeUse — Temporal subset to classify (3D input).** If X is a 3D, space-by-time-by-trial matrix, then this option will subset X along the time dimension. The input argument should be passed in as a vector of indices that indicate the time dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 2D, trial-by-feature matrix.

- **spaceUse — Spatial subset to classify (3D input).** If X is a 3D, space-by-time-by-trial matrix, then this option will subset X along the space dimension. The input argument should be passed in as a vector of indices that indicate the space dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 2D, trial-by-feature matrix.
- **featureUse — Temporal subset to classify (2D input).** If X is a 2D, trial-by-feature matrix, then this option will subset X along the features dimension. The input argument should be passed in as a vector of indices that indicate the feature dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 3D, space-by-time-by-trial matrix.
- **rngType — Random number generator (rng) specification.** If rngType is not entered or is empty, rng will be assigned here as {'shuffle', 'twister'}. The rngType input can be specified in the following ways:
 - Single acceptable rng specification input (e.g., 1, 'default', 'shuffle'); in these cases, the generator will be set to 'twister'.
 - Dual-argument specifications as either a 2-element cell array (e.g., {'shuffle', 'twister'}) or string array (e.g., ["shuffle", "twister"]).
 - rng struct as previously assigned by rngType = rng.
- **PCA — Principal Component Analysis specification.** Set principal component analysis on data matrix X. To retain components that explain a certain percentage of variance, enter a decimal value [0, 1). To retain a certain number of principal components, enter an integer greater or equal to 1. Default value is .99, which selects principal components that explain 99% of the variance. Enter 0 to disable PCA. PCA is computed along the feature dimension – that is, along the column dimension of the trial-by-feature matrix that is input to the classifier. Options:
 - Decimal between [0, 1) (default value 0.99): Include as many features as explain $N * 100$ percent of the variance of the data.
 - Integer greater than or equal to 1: Include the first N features.
 - 0: Do not perform PCA.
- **PCAIinFold — When PCA is computed.** This input specifies whether PCA is conducted on the training partition of each fold during cross validation, or if PCA is conducted once on the entire dataset prior to partitioning data for cross validation. Options:
 - true: Conduct PCA within each fold.

- `false` (default): Conduct PCA once across the entire data matrix X .
- **nFolds — Number of cross-validation folds.** Must be an integer greater than 1 and less than or equal to the number of trials. Default is 10.
- **classifier — Classifier used during cross validation.** Supported classifiers include support vector machine (SVM), linear discriminant analysis (LDA) and random forest (RF). Options:
 - `'LDA'` (default): Linear discriminant analysis.
 - `'SVM'`: Support vector machine. If using this classifier, the user must manually specify hyperparameter C (linear, rbf kernels) and γ (rbf kernel). If parameters are not known, use the `crossValidateMulti_opt` function to optimize SVM hyperparameters.
 - `'RF'`: Random forest.

Note for SVM classifications: If, when performing any SVM classification, 1) you receive an error message regarding an SVM function not being found, 2) the function hangs (no changes to the pop-up progress bar for several minutes), or 3) classifier performance is unexpectedly poor, you may still need to set up LIBSVM. See Chapter 2.2.3 for more information.

- **kernel — Decision function specification (SVM classifier).** This input is only relevant if the SVM classifier is selected. Options:
 - `'linear'`: Hyperparameter C
 - `'rbf'` (default): Hyperparameters γ (gamma) and C

For more information on the SVM classification kernels, see Hsu, Chang and Lin's 2003 paper, "A Practical Guide to Support Vector Classification".

- **gamma — SVM rbf kernel hyperparameter specification.** If SVM is selected as the classifier, and rbf is selected as the kernel, then γ must be manually set by the user.
- **C — SVM rbf and linear kernel hyperparameter specification.** If SVM is selected as the classifier, then C must be manually set by the user.
- **numTrees — Random Forest classifier hyperparameter.** This chooses the number of decision trees to grow. Default is 128.
- **minLeafSize — Random Forest classifier hyperparameter.** Specify the minimum number of observations per tree leaf. Default is 1.

- **permutations — Number of permutations during permutation testing.** If this value is set to 0, then permutation testing will be turned off. If it is set to an integer n greater than 0, then classification will be performed over n permutation iterations. Default value is 0 (off).

- - - Implementation notes - - -

This function repeats the following steps for each permutation:

- Select the first fold of training, test data (permutation testing will run only on this fold)
 - Permute the training labels
 - Train classifier on training data (with permuted labels)
 - Use the classifier to predict test data labels
 - Store the classification accuracy of this permutation
- **center — Data centering parameter.** Also known as mean centering. Setting this to any non-zero value will set the mean along the feature dimension to be 0. Setting to 0 turns it off. If PCA is performed, data centering is required; if the user selects a PCA calculation but 'center' is off, the function will issue a warning and turn centering on. Options:
 - true (default): Centering turned on
 - false: Centering turned off
 - **scale — Data scaling parameter.** Also known as data normalization. Setting this to a non-zero value to scales each feature to have unit variance prior to PCA. Setting it to 0 turns off data scaling. Options:
 - false (default): Scaling turned off
 - true: Scaling turned on

For more information on SVM hyperparameters, see Hsu et al. (2003), 'A Practical Guide to Support Vector Classification'. For more info on hyperparamters for random forest, see Matlab documentation for the `treeBagger()` class.²¹

Outputs

- **C — Classification output struct.** This output object contains all cross validation related information, including classification accuracy, confusion matrix, prediction results etc. The struct contains the following subfields:

²¹<https://www.mathworks.com/help/stats/treebagger.html>

- **CM**: Confusion matrix that summarizes the performance of the classification, in which rows represent actual labels and columns represent predicted labels. Element i, j represents the number of observations belonging to class i that the classifier labeled as belonging to class j .
- **accuracy**: Overall classification accuracy, computed as the number of correct predictions divided by the total number of predictions. Will be a value between 0 and 1.
- **predY**: Vector of predicted labels. Ordering of vector elements corresponds to the order of elements in input labels vector Y .
- **modelsConcat**: Struct containing the *nFold* models used during cross validation.
- **elapsedTime**: Runtime, in seconds.
- **pVal**: The p-value calculated using the permutation testing results. This is set to NaN if permutation testing is turned off.
- **permAccs**: Permutation testing accuracies. This field will be NaN if permutation testing is turned off.
- **classificationInfo**: This struct contains the specifications used during classification, including 'PCA', 'PCAIinFold', 'nFolds', 'classifier', and 'dataPartitionObj'.
- **dataPartitionObj**: This struct contains the train/test data partitions for cross validation (and a dev data partition if hyperparameter optimization is specified).

Example function calls

Here is a basic function call to `crossValidateMulti()`. The default setting, when unspecified, would be using LDA as the classifier, PCA set to explain 99% of the variance, `PCAIinFold` to be 0 (meaning PCA is conducted a single time on the entire dataset), and `nFold` as 10 to set the default number of cross validation folds to be 10.

```
M = Classification.crossValidateMulti(X, labels6);
M.classificationInfo
```

Here is another example call, this time where the classifier is set to RF (Random Forest), with the `minLeafSize` parameter set to 2.

```
%% Random Forest Hyperparameters: Leaf Size

M = Classification.crossValidateMulti(X , labels6, ...
    'classifier', 'RF', ...
    'PCA', 0.99, ...
```

```

        'minLeafSize', 2 ...
    );
    M.classificationInfo

```

A runnable script with example function calls is provided in the ExampleFunctionCalls folder: `example_Classification_crossValidateMulti.m`.

5.5 crossValidateMulti_opt()

Given a data matrix X and labels vector Y, this function will first conduct hyperparameter optimization, then conduct multiclass cross validation with an optimized classifier, and finally output a struct containing the classification accuracy, confusion matrix, and other related information. Other optional name-value parameters can be passed in to specify classification related options.

This function uses the helper functions (which can be found in the +Utils folder): `initInputParser()`, `subsetTrainTestMatrices()`, `setUserSpecifiedRng()`, `trainDevTestPart()`, `cvData()`, `trainDevGridSearch()`, `nestedCvGridSearch()`, `computeAccuracy()`.

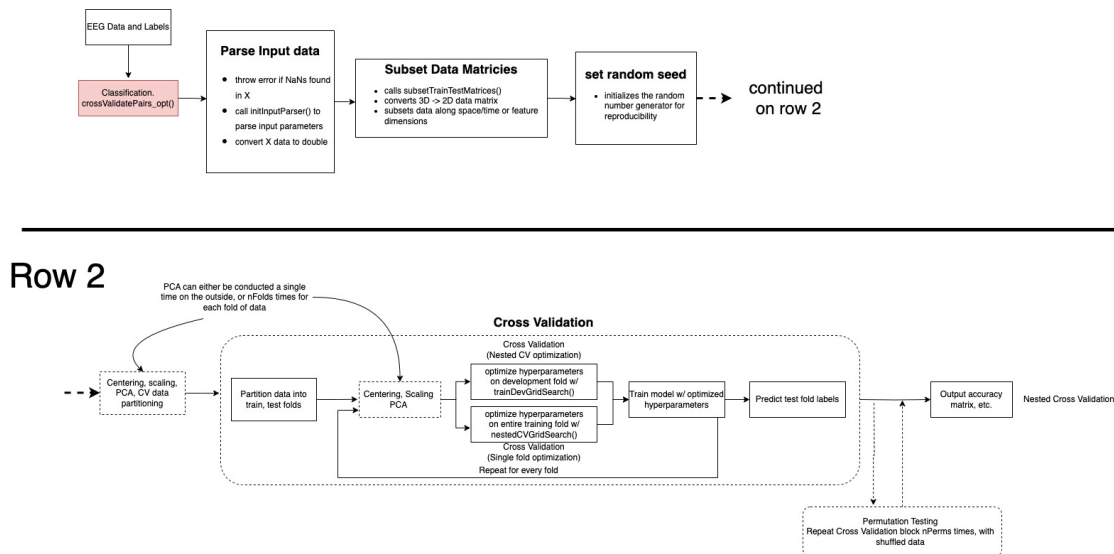


Figure 8: Overview of `crossValidateMulti_opt()` functionality.

Syntax

```
C = crossValidateMulti_opt(X, Y, varargin);
```

Required inputs

- **X — M/EEG data matrix.** The X matrix contains the response data. The X matrix can be passed into the function in the following shapes:

- A 3D sensor-by-time-by-trial matrix.
 - A 2D trial-by-feature matrix. For example, this matrix could contain response activations at a single time sample recorded from multiple sensors, or across time from a single sensor. In the function code, 2D data matrices are effectively treated as trial-by-time (single-sensor) matrices.
- **Y — Labels vector.** The Y vector contains the numeric labels corresponding to each trial in the M/EEG data matrix X. Both row and column vectors will be accepted. The length of Y must correspond to the size of X along the trial dimension.

Optional name-value inputs

- **timeUse — Temporal subset to classify (3D input).** If X is a 3D, space-by-time-by-trial matrix, then this option will subset X along the time dimension. The input argument should be passed in as a vector of indices that indicate the time dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 2D, trial-by-feature matrix.
- **spaceUse — Spatial subset to classify (3D input).** If X is a 3D, space-by-time-by-trial matrix, then this option will subset X along the space dimension. The input argument should be passed in as a vector of indices that indicate the space dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 2D, trial-by-feature matrix.
- **featureUse — Temporal subset to classify (2D input).** If X is a 2D, trial-by-feature matrix, then this option will subset X along the features dimension. The input argument should be passed in as a vector of indices that indicate the feature dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 3D, space-by-time-by-trial matrix.
- **rngType — Random number generator (rng) specification.** If rngType is not entered or is empty, rng will be assigned here as {'shuffle', 'twister'}. The rngType input can be specified in the following ways:
 - Single acceptable rng specification input (e.g., 1, 'default', 'shuffle'); in these cases, the generator will be set to 'twister'.
 - Dual-argument specifications as either a 2-element cell array (e.g., {'shuffle', 'twister'}) or string array (e.g., ["shuffle", "twister"]).
 - rng struct as previously assigned by rngType = rng.

- **PCA — Principal Component Analysis specification.** Set principal component analysis on data matrix X . To retain components that explain a certain percentage of variance, enter a decimal value $[0, 1)$. To retain a certain number of principal components, enter an integer greater or equal to 1. Default value is .99, which selects principal components that explain 99% of the variance. Enter 0 to disable PCA. PCA is computed along the feature dimension – that is, along the column dimension of the trial-by-feature matrix that is input to the classifier. Options:
 - Decimal between $[0, 1)$ (default value 0.99): Include as many features as explain $N * 100$ percent of the variance of the data.
 - Integer greater than or equal to 1: Include the first N features.
 - 0: Do not perform PCA.
- **PCAIinFold — When PCA is computed.** This input specifies whether PCA is conducted on the training partition of each fold during cross validation, or if PCA is conducted once on the entire dataset prior to partitioning data for cross validation. Options:
 - true: Conduct PCA within each fold.
 - false (default): Conduct PCA once across the entire data matrix X .
- **nFolds — Number of cross-validation folds.** Must be an integer greater than 1 and less than or equal to the number of trials. Default is 10. This parameter is only used if the 'nestedCV' option is set for 'optimization' parameter.
- **classifier — Classifier used during cross validation.** Currently, only SVM is supported for hyperparameter optimization:
 - 'SVM' (default)

Note for SVM classifications: If, when performing any SVM classification, 1) you receive an error message regarding an SVM function not being found, 2) the function hangs (no changes to the pop-up progress bar for several minutes), or 3) classifier performance is unexpectedly poor, you may still need to set up LIBSVM. See Chapter 2.2.3 for more information.
- **kernel — Decision function specification (SVM classifier).** This input is only relevant if the SVM classifier is selected. Options:
 - 'linear': Hyperparameter C
 - 'rbf' (default): Hyperparameters γ (gamma) and C

For more information on the SVM classification kernels, see Hsu, Chang and Lin's 2003 paper, "A Practical Guide to Support Vector Classification".

- **optimization — Optimization data specification.** This parameter controls whether optimization is conducted using a full *nFolds* cross validation on the training data or using a single development fold. Options:
 - 'singleFold' (default)
 - 'nestedCV'
- **trainDevSplit — Specification on fold split into training and development data.** This parameter is a 2 element vector which controls how each fold further split into a training and development data. For each fold, a (1/nFolds) fraction of the data becomes the test data, and a (1 - 1/nFolds) fraction of the data is used as the training/development partition. 'trainDevSplit' controls what fraction of the the training/development partition is used for training. Thus, for each fold, ((1 - 1/nFolds) * trainDevSplit) is used for training data, and ((1 - 1/nFolds) * (1-trainDevSplit)) is used for development data.
- **gammaSpace — Gamma search space.** Vector of *gamma* values to search over during hyperparameter optimization. *Gamma* is a hyperparameter of the rbf kernel for SVM classification. Default is 5 logarithmically spaced points between 10^{-5} and 10^5 .
- **cSpace — C search space.** Vector of *C* values to search over during hyperparameter optimization. See input argument *C* is a hyperparameter of both the rbf and linear kernels for SVM classification. Default is 5 logarithmically spaced points between 10^{-5} and 10^5 .
- **permutations — Number of permutations during permutation testing.** If this value is set to 0, then permutation testing will be turned off. If it is set to an integer *n* greater than 0, then classification will be performed over *n* permutation iterations. Default value is 0 (off).

- - - Implementation notes - - -

This function repeats the following steps for each permutation:

- Select the first fold of training, test data (permutation testing will run only on this fold)
- Permute the training labels
- Do hyperparameter optimization (using either the development fold or a nested cross validation on the training data, depending on which option is specified)
- Use the classifier to predict test data labels

- Store the classification accuracy of this permutation
- **center — Data centering parameter.** Also known as mean centering. Setting this to any non-zero value will set the mean along the feature dimension to be 0. Setting to 0 turns it off. If PCA is performed, data centering is required; if the user selects a PCA calculation but ‘center’ is off, the function will issue a warning and turn centering on. Options:
 - true (default): Centering turned on
 - false: Centering turned off
- **scale — Data scaling parameter.** Also known as data normalization. Setting this to a non-zero value to scales each feature to have unit variance prior to PCA. Setting it to 0 turns off data scaling. Options:
 - false (default): Scaling turned off
 - true: Scaling turned on

For more information on SVM hyperparameters, see Hsu et al. (2003), ‘A Practical Guide to Support Vector Classification’. For more info on hyperparameters for random forest, see Matlab documentation for the `treeBagger()` class.²²

Output

- **C — Classification output struct.** This output object contains all cross validation related information, including classification accuracy, confusion matrix, prediction results etc. The struct contains the following subfields:
 - **CM:** Confusion matrix that summarizes the performance of the classification, in which rows represent actual labels and columns represent predicted labels. Element i, j represents the number of observations belonging to class i that the classifier labeled as belonging to class j .
 - **C:** SVM hyperparameter optimized using grid search.
 - **gamma:** SVM hyperparameter optimized using grid search.
 - **accuracy:** Overall classification accuracy, computed as the number of correct predictions divided by the total number of predictions. Will be a value between 0 and 1.
 - **predY:** Vector of predicted labels. Ordering of vector elements corresponds to the order of elements in input labels vector Y.

²²<https://www.mathworks.com/help/stats/treebagger.html>

- **modelsConcat**: Struct containing the *nFold* models used during cross validation.
- **elapsedTime**: Runtime, in seconds.
- **pVal**: The p-value calculated using the permutation testing results. This is set to NaN if permutation testing is turned off.
- **permAccs**: Permutation testing accuracies. This field will be NaN if permutation testing is turned off.
- **classificationInfo**: This struct contains the specifications used during classification, including 'PCA', 'PCAIinFold', 'nFolds', 'classifier', and 'dataPartitionObj'.
- **dataPartitionObj**: This struct contains the train/test data partitions for cross validation (and a dev data partition if hyperparameter optimization is specified).

Example function calls

Here is a basic call to `crossValidateMulti_opt()`:

```
% X stores the data matrix, while labels6 stores the labels

M = Classification.crossValidateMulti_opt(X, labels6);
M.classificationInfo
```

Here is another function call, this time specifying SVM as the classifier, and using the linear kernel:

```
M = Classification.crossValidateMulti_opt(X, labels6, ...
'classifier', 'SVM', ...
'PCA', 0.99, ...
'kernel', 'linear'...
);

M.classificationInfo
```

A runnable script with example function calls is provided in the `ExampleFunctionCalls` folder: `example_Classification_crossValidateMulti_opt.m`.

5.6 crossValidatePairs()

This function averages trials in the data matrix *X* on a per-label basis (as defined by elements of *Y*) in groups of `groupSize` trials. In other words, the function averages groups of trials belonging to the same category, where the number of trials averaged in each group is specified by the variable `groupSize`.

- The user can also optionally enter a vector of participant identifiers P, in which case trial averaging will additionally be performed on a per-participant basis.
- The function takes in optional name-value pairs to specify handling of remainder trials, whether to shuffle the data after averaging (retaining the mapping between trials and labels), and to set the random number generator.

If the user wishes to shuffle the ordering of data (while preserving labeling of data observations) prior to averaging, they should call the `shuffleData()` function prior to calling this function.

This function uses the helper functions (which can be found in the +Utils folder): `initInputParser()`, `subsetTrainTestMatrices()`, `setUserSpecifiedRng()`, `initPairwiseCellMat()`, `trainDevTestPart()`, `cvData()`, `fitModel()`, `modelPredict()`, `computeAccuracy()`, `permTestPVal()`, `pValMat()`.

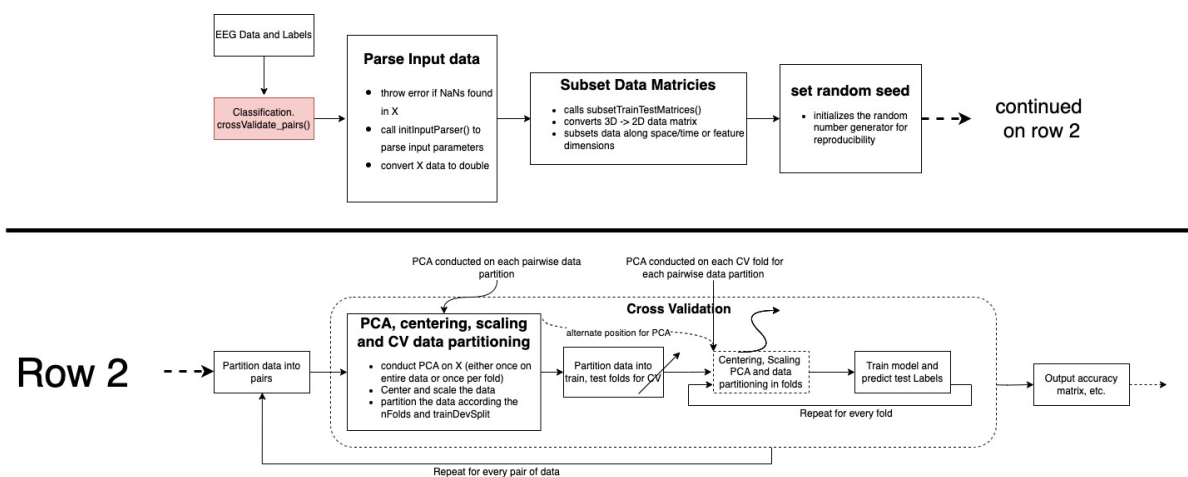


Figure 9: Overview of `crossValidatePairs()` functionality.

Syntax

```
C = Classification.crossValidatePairs(X, Y, varargin);
```

Required inputs

- **X — M/EEG data matrix.** The X matrix contains the response data. The X matrix can be passed into the function in the following shapes:
 - A 3D sensor-by-time-by-trial matrix.
 - A 2D trial-by-feature matrix. For example, this matrix could contain response activations at a single time sample recorded from multiple sensors, or across time from a single sensor. In the function code, 2D data matrices are effectively treated as trial-by-time (single-sensor) matrices.

- **Y — Labels vector.** The Y vector contains the numeric labels corresponding to each trial in the M/EEG data matrix X. Both row and column vectors will be accepted. The length of Y must correspond to the size of X along the trial dimension.

Optional name-value inputs

- **timeUse — Temporal subset to classify (3D input).** If X is a 3D, space-by-time-by-trial matrix, then this option will subset X along the time dimension. The input argument should be passed in as a vector of indices that indicate the time dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 2D, trial-by-feature matrix.
- **spaceUse — Spatial subset to classify (3D input).** If X is a 3D, space-by-time-by-trial matrix, then this option will subset X along the space dimension. The input argument should be passed in as a vector of indices that indicate the space dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 2D, trial-by-feature matrix.
- **featureUse — Temporal subset to classify (2D input).** If X is a 2D, trial-by-feature matrix, then this option will subset X along the features dimension. The input argument should be passed in as a vector of indices that indicate the feature dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 3D, space-by-time-by-trial matrix.
- **rngType — Random number generator (rng) specification.** If rngType is not entered or is empty, rng will be assigned here as {'shuffle', 'twister'}. The rngType input can be specified in the following ways:
 - Single acceptable rng specification input (e.g., 1, 'default', 'shuffle'); in these cases, the generator will be set to 'twister'.
 - Dual-argument specifications as either a 2-element cell array (e.g., {'shuffle', 'twister'}) or string array (e.g., ["shuffle", "twister"]).
 - rng struct as previously assigned by rngType = rng.
- **PCA — Principal Component Analysis specification.** Set principal component analysis on data matrix X. To retain components that explain a certain percentage of variance, enter a decimal value [0, 1). To retain a certain number of principal components, enter an integer greater or equal to 1. Default value is .99, which selects principal components that explain 99% of the variance. Enter 0 to disable PCA. PCA is computed along the feature dimension – that is, along the column dimension of the trial-by-feature matrix that is input to the classifier. Options:

- Decimal between $[0, 1)$ (default value 0.99): Include as many features as explain $N * 100$ percent of the variance of the data.
 - Integer greater than or equal to 1: Include the first N features.
 - 0: Do not perform PCA.
- **PCAIinFold — When PCA is computed.** This input specifies whether PCA is conducted on the training partition of each fold during cross validation, or if PCA is conducted once on the entire dataset prior to partitioning data for cross validation. Options:
 - `true`: Conduct PCA within each fold.
 - `false` (default): Conduct PCA once across the entire data matrix X .
 - **nFolds — Number of cross-validation folds.** Must be an integer greater than 1 and less than or equal to the number of trials. Default is 10.
 - **classifier — Classifier used during cross validation.** Supported classifiers include support vector machine (SVM), linear discriminant analysis (LDA) and random forest (RF). Options:
 - `'LDA'` (default): Linear discriminant analysis.
 - `'SVM'`: Support vector machine. If using this classifier, the user must manually specify hyperparameter C (linear, rbf kernels) and γ (rbf kernel). If parameters are not known, use the `crossValidateMulti_opt` function to optimize SVM hyperparameters.
 - `'RF'`: Random forest.

Note for SVM classifications: If, when performing any SVM classification, 1) you receive an error message regarding an SVM function not being found, 2) the function hangs (no changes to the pop-up progress bar for several minutes), or 3) classifier performance is unexpectedly poor, you may still need to set up LIBSVM. See Chapter 2.2.3 for more information.

- **kernel — Decision function specification (SVM classifier).** This input is only relevant if the SVM classifier is selected. Options:
 - `'linear'`: Hyperparameter C
 - `'rbf'` (default): Hyperparameters γ (gamma) and C

For more information on the SVM classification kernels, see Hsu, Chang and Lin's 2003 paper, "A Practical Guide to Support Vector Classification".

- **C — Hyperparameter rbf, linear kernels for SVM classification.** If SVM is selected as the classifier, then C must be manually set by the user. It is recommended for the user to run the corresponding ‘_opt’ function to determine a suitable C value.
- **gamma — Hyperparameter of the rbf kernel for SVM classification.** If SVM is selected as the classifier, and rbf is selected as the kernel, then gamma must be manually set by the user. It is recommended for the user to run the corresponding ‘_opt’ function to determine a suitable gamma value.
- **numTrees — Hyperparameter of the random forest classifier.** This chooses the number of decision trees to grow. Default is 128.
- **minLeafSize — Hyperparameter of the random forest classifier.** Choose the minimum number of observations per tree leaf. Default is 1.
- **permutations — Number of permutations during permutation testing.** If this value is set to 0, then permutation testing will be turned off. If it is set to an integer n greater than 0, then classification will be performed over n permutation iterations. Default value is 0 (off).

- - - Implementation notes - - -

This function repeats the following steps for each decision boundary:

- Select the first fold of training, test data (permutation testing will run only on this fold)
- Select the data from this fold that represents the two classes of interest
- For each permutation, repeat the following steps:
 - * Permute the training labels
 - * Train a two-class classifier on training data (with permuted labels)
 - * Use the two-class classifier to predict test data labels
 - * Store the classification accuracy of this permutation and decision boundary
- **center — Data centering parameter.** Also known as mean centering. Setting this to any non-zero value will set the mean along the feature dimension to be 0. Setting to 0 turns it off. If PCA is performed, data centering is required; if the user selects a PCA calculation but ‘center’ is off, the function will issue a warning and turn centering on. Options:
 - true (default): Centering turned on
 - false: Centering turned off

- **scale — Data scaling parameter.** Also known as data normalization. Setting this to a non-zero value to scales each feature to have unit variance prior to PCA. Setting it to 0 turns off data scaling. Options:
 - `false` (default): Scaling turned off
 - `true`: Scaling turned on

Outputs

- **C — Pairwise classification output struct.** This output object contains all pairwise cross validation related information, including classification accuracy, confusion matrix, prediction results for each decision boundary. The struct contains the following subfields:
 - **pairwiseInfo**: a cell matrix of structs, symmetrical along the diagonal. Each struct contains the following subfields, for each class:
 - * **CM** - Confusion matrix that summarizes the performance of the classification, in which rows represent actual labels and columns represent predicted labels. Element *i,j* represents the number of observations of class *i* that the classifier labeled as belonging to class *j*.
 - * **classBoundary** - The pair of classes corresponding to the current index (i.e. class 1 vs class 4).
 - * **accuracy** - Classification accuracy for the given class boundary
 - actualLabels** - Actual class labels
 - predictions** - Predicted class labels by the trained model
 - **AM**: Accuracy matrix where each off-diagonal element shows the accuracy for distinguishing one class from another. Since we don't compare classes to themselves, the diagonal will contain NaNs.
 - **pValMat**: If permutation testing is specified, this output contains a matrix of the percentile value of the found value among the permutation test values. Similar to the accuracy matrix, the diagonal will contain NaNs.
 - **permAccMat**: Permutation testing accuracies. This field will be NaN if permutation testing is not specified. The first two dimensions represent pairwise classes, while the third dimension represent permutation.
 - **modelsConcat**: Struct containing the models trained during cross validation.
 - **elapsedTime**: Runtime, in seconds.
 - **classificationInfo**: This struct contains the specifications used during classification, including 'PCA', 'PCAIinFold', 'nFolds', 'classifier', and 'dataPartitionObj'.

- **dataPartitionObj**: This struct contains the train/test data partitions for cross validation (and a dev data partition if hyperparameter optimization is specified).
- **avgAccuracy**: Average classification accuracy across all class boundaries.

Example function calls

Here is a basic call to `crossValidatePairs()` with an `rngType` specified for reproducibility:

```
M = Classification.crossValidatePairs(X, labels6, 'rngType', 3);
M.classificationInfo
```

Here is another function call, with `PCAIinFold` set to true, such that principal component analysis is conducted separately within every fold:

```
M = Classification.crossValidatePairs(X, labels6, 'PCA', .99, ...
'classifier', 'LDA', 'PCAIinFold', 1);
M.classificationInfo
```

A runnable script with example function calls is provided in the `ExampleFunctionCalls` folder: `example_Classification_crossValidatePairs.m`.

5.7 crossValidatePairs_opt()

This function averages trials in the data matrix `X` on a per-label basis (as defined by elements of `Y`) in groups of `groupSize` trials. In other words, the function averages groups of trials belonging to the same category, where the number of trials averaged in each group is specified by the variable `groupSize`.

- The user can also optionally enter a vector of participant identifiers `P`, in which case trial averaging will additionally be performed on a per-participant basis.
- The function takes in optional name-value pairs to specify handling of remainder trials, whether to shuffle the data after averaging (retaining the mapping between trials and labels), and to set the random number generator.

If the user wishes to shuffle the ordering of data (while preserving labeling of data observations) prior to averaging, they should call the `shuffleData()` function prior to calling this function.

This function uses the helper functions (which can be found in the `+Utils` folder): `initInputParser()`, `subsetTrainTestMatrices()`, `setUserSpecifiedRng()`, `trainDevTestPart()`, `cvData()`, `initPairwiseCellMat()`,

classTuple2Nchoose2Ind(), nestedCvGridSearch(), trainDevGridSearch(),
fitModel(), modelPredict(), permTestPVal().

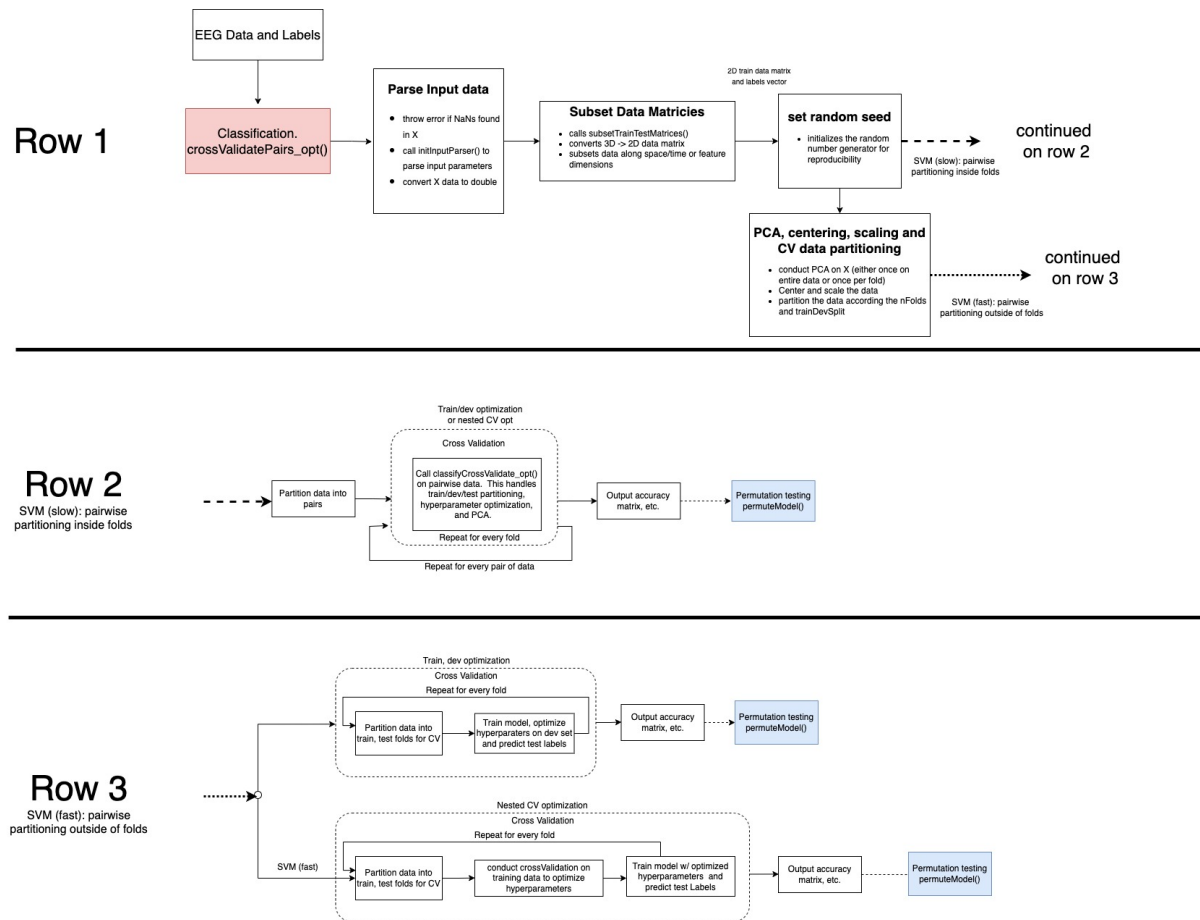


Figure 10: Overview of `crossValidatePairs_opt()` functionality.

Syntax

```
C = Classification.crossValidatePairs_opt(X, Y, varargin);
```

Required inputs

- **X — M/EEG data matrix.** The X matrix contains the response data. The X matrix can be passed into the function in the following shapes:
 - A 3D sensor-by-time-by-trial matrix.
 - A 2D trial-by-feature matrix. For example, this matrix could contain response activations at a single time sample recorded from multiple sensors, or across time from a single sensor. In the function code, 2D data matrices are effectively treated as trial-by-time (single-sensor) matrices.
- **Y — Labels vector.** The Y vector contains the numeric labels corresponding to each trial in the M/EEG data matrix X. Both row and column vectors will be accepted. The length of Y must correspond to the size of X along the trial dimension.

- **groupSize** — **Number of single trials to be averaged in each group.** For example, a groupSize value of 5 would mean that groups of 5 trials are averaged, *not* that the available trials are partitioned into 5 groups total.

Optional inputs

- **P** — **Participant vector.** The optional input P specifies the participant identifier of every trial of data. The length of P must correspond to the length of Y and the length of the trial dimension of X. If P is not entered or is empty, the function will return NaN as randomized P. P can be a numeric vector, string array, or cell array.

Optional name-value inputs

- **timeUse** — **Temporal subset to classify (3D input).** If X is a 3D, space-by-time-by-trial matrix, then this option will subset X along the time dimension. The input argument should be passed in as a vector of indices that indicate the time dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 2D, trial-by-feature matrix.
- **spaceUse** — **Spatial subset to classify (3D input).** If X is a 3D, space-by-time-by-trial matrix, then this option will subset X along the space dimension. The input argument should be passed in as a vector of indices that indicate the space dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 2D, trial-by-feature matrix.
- **featureUse** — **Temporal subset to classify (2D input).** If X is a 2D, trial-by-feature matrix, then this option will subset X along the features dimension. The input argument should be passed in as a vector of indices that indicate the feature dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 3D, space-by-time-by-trial matrix.
- **rngType** — **Random number generator (rng) specification.** If rngType is not entered or is empty, rng will be assigned here as {'shuffle', 'twister'}. The rngType input can be specified in the following ways:
 - Single acceptable rng specification input (e.g., 1, 'default', 'shuffle'); in these cases, the generator will be set to 'twister'.
 - Dual-argument specifications as either a 2-element cell array (e.g., {'shuffle', 'twister'}) or string array (e.g., ["shuffle", "twister"]).
 - rng struct as previously assigned by rngType = rng.

- **PCA — Principal Component Analysis specification.** Set principal component analysis on data matrix X . To retain components that explain a certain percentage of variance, enter a decimal value $[0, 1)$. To retain a certain number of principal components, enter an integer greater or equal to 1. Default value is .99, which selects principal components that explain 99% of the variance. Enter 0 to disable PCA. PCA is computed along the feature dimension – that is, along the column dimension of the trial-by-feature matrix that is input to the classifier. Options:
 - Decimal between $[0, 1)$ (default value 0.99): Include as many features as explain $N * 100$ percent of the variance of the data.
 - Integer greater than or equal to 1: Include the first N features.
 - 0: Do not perform PCA.
- **PCAIinFold — When PCA is computed.** This input specifies whether PCA is conducted on the training partition of each fold during cross validation, or if PCA is conducted once on the entire dataset prior to partitioning data for cross validation. Options:
 - `true`: Conduct PCA within each fold.
 - `false` (default): Conduct PCA once across the entire data matrix X .
- **nFolds — Number of cross-validation folds.** Must be an integer greater than 1 and less than or equal to the number of trials. Default is 10.
- **classifier — Classifier used during cross validation.** Currently, only SVM is supported for hyperparameter optimization:
 - `'SVM'` (default)

Note for SVM classifications: If, when performing any SVM classification, 1) you receive an error message regarding an SVM function not being found, 2) the function hangs (no changes to the pop-up progress bar for several minutes), or 3) classifier performance is unexpectedly poor, you may still need to set up LIBSVM. See Chapter 2.2.3 for more information.

- **kernel — Decision function specification (SVM classifier).** This input is only relevant if the SVM classifier is selected. Options:
 - `'linear'`: Hyperparameter C
 - `'rbf'` (default): Hyperparameters γ (gamma) and C

For more information on the SVM classification kernels, see Hsu, Chang and Lin's 2003 paper, "A Practical Guide to Support Vector Classification".

- **optimization — Optimization data specification.** This parameter controls whether optimization is conducted using a full *nFolds* cross validation on the training data or using a single development fold. Options:
 - 'singleFold' (default)
 - 'nestedCV'
- **trainDevSplit — Specification on fold split into training and development data.** This parameter is a 2 element vector which controls how each fold further split into a training and development data. For each fold, a $(1/nFolds)$ fraction of the data becomes the test data, and a $(1 - 1/nFolds)$ fraction of the data is used as the training/development partition. 'trainDevSplit' controls what fraction of the the training/development partition is used for training. Thus, for each fold, $((1 - 1/nFolds) * \text{trainDevSplit})$ is used for training data, and $((1 - 1/nFolds) * (1 - \text{trainDevSplit}))$ is used for development data.
- **gammaSpace — Gamma search space.** Vector of *gamma* values to search over during hyperparameter optimization. *Gamma* is a hyperparameter of the rbf kernel for SVM classification. Default is 5 logarithmically spaced points between 10^{-5} and 10^5 .
- **cSpace — C search space.** Vector of *C* values to search over during hyperparameter optimization. See input argument *C* is a hyperparameter of both the rbf and linear kernels for SVM classification. Default is 5 logarithmically spaced points between 10^{-5} and 10^5 .
- **permutations — Number of permutations during permutation testing.** If this value is set to 0, then permutation testing will be turned off. If it is set to an integer *n* greater than 0, then classification will be performed over *n* permutation iterations. Default value is 0 (off).

- - - Implementation notes - - -

This function repeats the following steps for each permutation:

- Select the first fold of training, test data (permutation testing will run only on this fold)
- Permute the training labels
- Do hyperparameter optimization (using either the development fold or a nested cross validation on the training data, depending on which option is specified)
- Train classifier on training data (with permuted labels)
- Use the classifier to predict test data labels

- Use the helper function `decValues2PairwiseAcc()` to convert the LIBSVM decision values to pairwise permutation testing accuracies
- **center — Data centering parameter.** Also known as mean centering. Setting this to any non-zero value will set the mean along the feature dimension to be 0. Setting to 0 turns it off. If PCA is performed, data centering is required; if the user selects a PCA calculation but ‘center’ is off, the function will issue a warning and turn centering on. Options:
 - `true` (default): Centering turned on
 - `false`: Centering turned off
- **scale — Data scaling parameter.** Also known as data normalization. Setting this to a non-zero value to scales each feature to have unit variance prior to PCA. Setting it to 0 turns off data scaling. Options:
 - `false` (default): Scaling turned off
 - `true`: Scaling turned on

Outputs

- **C — Pairwise classification output struct.** This output object contains all pairwise cross validation related information, including classification accuracy, confusion matrix, prediction results for each decision boundary. The struct contains the following subfields:
 - **pairwiseInfo:** a cell matrix of structs, symmetrical along the diagonal. Each struct contains the following subfields, for each class:
 - * **CM** - Confusion matrix that summarizes the performance of the classification, in which rows represent actual labels and columns represent predicted labels. Element *i,j* represents the number of observations of class *i* that the classifier labeled as belonging to class *j*.
 - * **classBoundary** - The pair of classes corresponding to the current index (i.e. class 1 vs class 4).
 - * **accuracy** - Classification accuracy for the given class boundary
 - * **actualLabels** - Actual class labels
 - * **predictions** - Predicted class labels by the trained model
 - **AM:** Accuracy matrix where each off-diagonal element shows the accuracy for distinguishing one class from another. Since we don’t compare classes to themselves, the diagonal will contain NaNs.

- **pValMat**: If permutation testing is specified, this output contains a matrix of the percentile value of the found value among the permutation test values. Similar to the accuracy matrix, the diagonal will contain NaNs.
- **permAccMat**: Permutation testing accuracies. This field will be NaN if permutation testing is not specified. The first two dimensions represent pairwise classes, while the third dimension represent permutation.
- **modelsConcat**: Struct containing the models trained during cross validation.
- **elapsedTime**: Runtime, in seconds.
- **classificationInfo**: This struct contains the specifications used during classification, including 'PCA', 'PCAIinFold', 'nFolds', 'classifier', and 'dataPartitionObj'.
- **dataPartitionObj**: This struct contains the train/test data partitions for cross validation (and a dev data partition if hyperparameter optimization is specified).
- **avgAccuracy**: Average classification accuracy across all class boundaries.

Example function calls

Here is a function call to `crossValidatePairs_opt()` with a that subset channels 94-100 for analysis:

```
M = Classification.crossValidatePairs_opt(X, labels6, 'PCA', .99, ...
    'spaceUse', 94:100);
```

Here is another call that uses the SVM classifier with the radial basis function (rbf) kernel:

```
M = Classification.crossValidatePairs_opt(X, labels6, ...
    'classifier', 'SVM', 'PCA', 0.99, 'kernel', 'rbf');
M.classificationInfo
```

A runnable script with example function calls is provided in the ExampleFunctionCalls folder: `example_Classification_crossValidatePairs_opt.m`.

5.8 trainMulti()

Given a data matrix X and labels vector Y, this function trains a classification model using the data, then outputs this model into a struct. This struct can be passed into the toolbox function `predict()` to predict the labels of future trials.

This function uses the helper functions (which can be found in the +Utils folder): `initInputParser()`, `checkInputDataShape()`,

verifySVMParameters(), subsetTrainTestMatrices(), setUserSpecifiedRng(), processTrainDevTestSplit(), trainDevTestPart(), cvData(), fitModel().

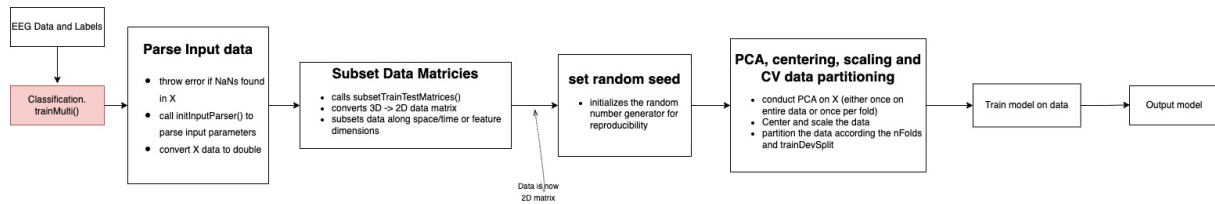


Figure 11: Overview of trainMulti() functionality.

Syntax

```
[M, permTestData] = trainMulti(X, Y, varargin);
```

Required inputs

- **X — M/EEG data matrix.** The X matrix contains the response data. The X matrix can be passed into the function in the following shapes:
 - A 3D sensor-by-time-by-trial matrix.
 - A 2D trial-by-feature matrix. For example, this matrix could contain response activations at a single time sample recorded from multiple sensors, or across time from a single sensor. In the function code, 2D data matrices are effectively treated as trial-by-time (single-sensor) matrices.
- **Y — Labels vector.** The Y vector contains the numeric labels corresponding to each trial in the M/EEG data matrix X. Both row and column vectors will be accepted. The length of Y must correspond to the size of X along the trial dimension.

Optional name-value inputs

- **timeUse — Temporal subset to classify (3D input).** If X is a 3D, space-by-time-by-trial matrix, then this option will subset X along the time dimension. The input argument should be passed in as a vector of indices that indicate the time dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 2D, trial-by-feature matrix.
- **spaceUse — Spatial subset to classify (3D input).** If X is a 3D, space-by-time-by-trial matrix, then this option will subset X along the space dimension. The input argument should be passed in as a vector of indices that indicate the space dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 2D, trial-by-feature matrix.

- **featureUse — Temporal subset to classify (2D input).** If X is a 2D, trial-by-feature matrix, then this option will subset X along the features dimension. The input argument should be passed in as a vector of indices that indicate the feature dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 3D, space-by-time-by-trial matrix.
- **rngType — Random number generator (rng) specification.** If `rngType` is not entered or is empty, `rng` will be assigned here as `{'shuffle', 'twister'}`. The `rngType` input can be specified in the following ways:
 - Single acceptable `rng` specification input (e.g., `1`, `'default'`, `'shuffle'`); in these cases, the generator will be set to `'twister'`.
 - Dual-argument specifications as either a 2-element cell array (e.g., `{'shuffle', 'twister'}`) or string array (e.g., `["shuffle", "twister"]`).
 - `rng` struct as previously assigned by `rngType = rng`.
- **PCA — Principal Component Analysis specification.** Set principal component analysis on data matrix X . To retain components that explain a certain percentage of variance, enter a decimal value $[0, 1)$. To retain a certain number of principal components, enter an integer greater or equal to 1. Default value is .99, which selects principal components that explain 99% of the variance. Enter 0 to disable PCA. PCA is computed along the feature dimension – that is, along the column dimension of the trial-by-feature matrix that is input to the classifier. Options:
 - Decimal between $[0, 1)$ (default value 0.99): Include as many features as explain $N * 100$ percent of the variance of the data.
 - Integer greater than or equal to 1: Include the first N features.
 - 0: Do not perform PCA.
- **nFolds — Number of cross-validation folds.** Must be an integer greater than 1 and less than or equal to the number of trials. Default is 10.
- **classifier — Classifier used during cross validation.** Supported classifiers include support vector machine (SVM), linear discriminant analysis (LDA) and random forest (RF). Options:
 - `'LDA'` (default): Linear discriminant analysis.
 - `'SVM'`: Support vector machine. If using this classifier, the user must manually specify hyperparameter `C` (linear, rbf kernels) and `gamma` (rbf kernel). If parameters are not known, use functions with `_opt` names to optimize SVM hyperparameters.

- 'RF': Random forest.

Note for SVM classifications: If, when performing any SVM classification, 1) you receive an error message regarding an SVM function not being found, 2) the function hangs (no changes to the pop-up progress bar for several minutes), or 3) classifier performance is unexpectedly poor, you may still need to set up LIBSVM. See Chapter 2.2.3 for more information.

- **kernel — Decision function specification (SVM classifier).** This input is only relevant if the SVM classifier is selected. Options:

- 'linear': Hyperparameter C
- 'rbf' (default): Hyperparameters γ (gamma) and C

For more information on the SVM classification kernels, see Hsu, Chang and Lin's 2003 paper, "A Practical Guide to Support Vector Classification".

- **gamma — SVM rbf kernel hyperparameter specification.** If SVM is selected as the classifier, and rbf is selected as the kernel, then gamma must be manually set by the user.
- **C — SVM rbf and linear kernel hyperparameter specification.** If SVM is selected as the classifier, then C must be manually set by the user.
- **numTrees — Random Forest classifier hyperparameter.** This chooses the number of decision trees to grow. Default is 128.
- **minLeafSize — Random Forest classifier hyperparameter.** Specify the minimum number of observations per tree leaf. Default is 1.
- **permutations — Number of permutations during permutation testing.** If this value is set to 0, then permutation testing will be turned off. If it is set to an integer n greater than 0, then classification will be performed over n permutation iterations. Default value is 0 (off).

- - - Implementation notes - - -

The permutation testing for this function is carried out by the predict() function, consistent with the steps described in the crossValidateMulti() function. Please refer to the permutations section in the crossValidateMulti() function documentation or the code docstring to learn more.

- **center — Data centering parameter.** Also known as mean centering. Setting this to any non-zero value will set the mean along the feature dimension to be 0. Setting to 0 turns it off. If PCA is performed, data centering is required; if the user selects a PCA calculation but 'center' is off, the function will issue a warning and turn centering on. Options:

- true (default): Centering turned on
- false: Centering turned off
- **scale — Data scaling parameter.** Also known as data normalization. Setting this to a non-zero value to scales each feature to have unit variance prior to PCA. Setting it to 0 turns off data scaling. Options:
 - false (default): Scaling turned off
 - true: Scaling turned on

For more information on SVM hyperparameters, see Hsu et al. (2003), ‘A Practical Guide to Support Vector Classification’. For more info on hyperparamters for random forest, see Matlab documentation for the `treeBagger()` class.²³

Outputs

- **M — Classification output to be passed into predict().** The struct contains the following subfields:
 - **classificationInfo:** Additional parameters/info for classification.
 - **mdl:** Classification model which is used in `predict()` to classify labels of new test data.
 - **classifier:** Classifier selected for training.
 - **functionName:** The name of the current function in string format.
 - **cvDataObj:** Object containing data and labels after PCA.
 - **permutations:** Please see 'permutations' section in the input arguments.
 - **ip:** Input parser object for this function.
 - **elapsedTime:** Time elapsed for training current model in seconds. Could be used to gauge permutation testing duration.
 - **scale:** Please see section for 'scale' input argument.
- **permTestData — Permutation testing data to be passed into predict().**

A runnable script with example function calls is provided in the `ExampleFunctionCalls` folder: `example_Classification_trainMulti.m`.

²³<https://www.mathworks.com/help/stats/treebagger.html>

5.9 trainMulti_opt()

Given a data matrix X and labels vector Y , this function will split the data into pairs of classes, optimize the classifier hyperparameters, then conduct cross validation. Then, an output struct will be passed out containing the classification accuracies, confusion matrices, and other information for each pair of labels. Optional name-value parameters can be passed in to specify classification related options.

Currently, the only classifier compatible w/ this function is SVM. Optimization is done via a grid search over the values specified in the `gammaSpace` and `cSpace` input parameters.

This function uses the helper functions (which can be found in the `+Utils` folder): `initInputParser()`, `subsetTrainTestMatrices()`, `setUserSpecifiedRng()`, `trainDevTestPart()`, `cvData()`, `trainDevGridSearch()`, `nestedCvGridSearch()`, `fitModel()`.

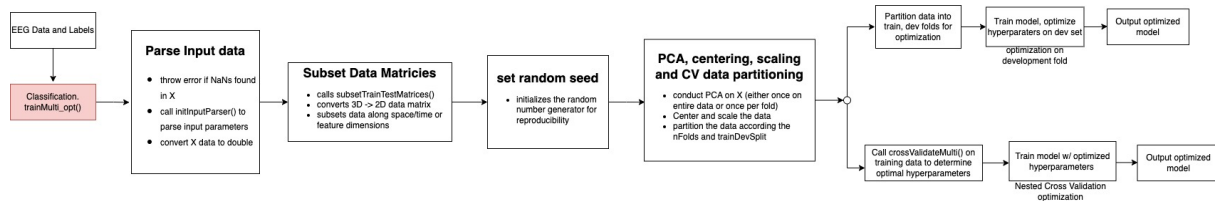


Figure 12: Overview of `trainMulti_opt()` functionality.

Syntax

```
[M, permTestData] = trainMulti_opt(X, Y, varargin);
```

Required inputs

- **X — M/EEG data matrix.** The X matrix contains the response data. The X matrix can be passed into the function in the following shapes:
 - A 3D sensor-by-time-by-trial matrix.
 - A 2D trial-by-feature matrix. For example, this matrix could contain response activations at a single time sample recorded from multiple sensors, or across time from a single sensor. In the function code, 2D data matrices are effectively treated as trial-by-time (single-sensor) matrices.
- **Y — Labels vector.** The Y vector contains the numeric labels corresponding to each trial in the M/EEG data matrix X . Both row and column vectors will be accepted. The length of Y must correspond to the size of X along the trial dimension.

Optional name-value inputs

- **timeUse — Temporal subset to classify (3D input).** If X is a 3D, space-by-time-by-trial matrix, then this option will subset X along the time dimension. The input argument should be passed in as a vector of indices that indicate the time dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 2D, trial-by-feature matrix.
- **spaceUse — Spatial subset to classify (3D input).** If X is a 3D, space-by-time-by-trial matrix, then this option will subset X along the space dimension. The input argument should be passed in as a vector of indices that indicate the space dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 2D, trial-by-feature matrix.
- **featureUse — Temporal subset to classify (2D input).** If X is a 2D, trial-by-feature matrix, then this option will subset X along the features dimension. The input argument should be passed in as a vector of indices that indicate the feature dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 3D, space-by-time-by-trial matrix.
- **rngType — Random number generator (rng) specification.** If rngType is not entered or is empty, rng will be assigned here as {'shuffle', 'twister'}. The rngType input can be specified in the following ways:
 - Single acceptable rng specification input (e.g., 1, 'default', 'shuffle'); in these cases, the generator will be set to 'twister'.
 - Dual-argument specifications as either a 2-element cell array (e.g., {'shuffle', 'twister'}) or string array (e.g., ["shuffle", "twister"]).
 - rng struct as previously assigned by rngType = rng.
- **PCA — Principal Component Analysis specification.** Set principal component analysis on data matrix X. To retain components that explain a certain percentage of variance, enter a decimal value [0, 1). To retain a certain number of principal components, enter an integer greater or equal to 1. Default value is .99, which selects principal components that explain 99% of the variance. Enter 0 to disable PCA. PCA is computed along the feature dimension – that is, along the column dimension of the trial-by-feature matrix that is input to the classifier. Options:
 - Decimal between [0, 1) (default value 0.99): Include as many features as explain $N * 100$ percent of the variance of the data.
 - Integer greater than or equal to 1: Include the first N features.
 - 0: Do not perform PCA.

- **nFolds_opt — Number of folds in cross validation.** Must be an integer greater than 2 or less than or equal to the number of trials. Default is 10.
- **classifier — Classifier used during cross validation.** Currently, only SVM is supported for hyperparameter optimization:

- 'SVM' (default)

Note for SVM classifications: If, when performing any SVM classification, 1) you receive an error message regarding an SVM function not being found, 2) the function hangs (no changes to the pop-up progress bar for several minutes), or 3) classifier performance is unexpectedly poor, you may still need to set up LIBSVM. See Chapter 2.2.3 for more information.

- **kernel — Decision function specification (SVM classifier).** This input is only relevant if the SVM classifier is selected. Options:

- 'linear': Hyperparameter C
 - 'rbf' (default): Hyperparameters γ (gamma) and C

For more information on the SVM classification kernels, see Hsu, Chang and Lin's 2003 paper, "A Practical Guide to Support Vector Classification".

- **optimization — Optimization data specification.** This parameter controls whether optimization is conducted using a full *nFolds* cross validation on the training data or using a single development fold. Options:

- 'singleFold' (default)
 - 'nestedCV'

- **trainDevSplit — Specification on fold split into training and development data.** This parameter is a 2 element vector which controls how each fold further split into a training and development data. For each fold, a $(1/nFolds)$ fraction of the data becomes the test data, and a $(1 - 1/nFolds)$ fraction of the data is used as the training/development partition. 'trainDevSplit' controls what fraction of the the training/development partition is used for training. Thus, for each fold, $((1 - 1/nFolds) * \text{trainDevSplit})$ is used for training data, and $((1 - 1/nFolds) * (1 - \text{trainDevSplit}))$ is used for development data.
- **gammaSpace — Gamma search space.** Vector of *gamma* values to search over during hyperparameter optimization. *Gamma* is a hyperparameter of the rbf kernel for SVM classification. Default is 5 logarithmically spaced points between 10^{-5} and 10^5 .

- **cSpace — C search space.** Vector of C values to search over during hyperparameter optimization. See input argument C is a hyperparameter of both the rbf and linear kernels for SVM classification. Default is 5 logarithmically spaced points between 10^{-5} and 10^5 .
- **permutations — Number of permutations during permutation testing.** If this value is set to 0, then permutation testing will be turned off. If it is set to an integer n greater than 0, then classification will be performed over n permutation iterations. Default value is 0 (off).

- - - Implementation notes - - -

The permutation testing for this function is carried out by the `predict()` function, consistent with the steps described in the `crossValidateMulti_opt()` function. Please refer to the permutations section in the `crossValidateMulti_opt()` function documentation or the code docstring to learn more.

- **center — Data centering parameter.** Also known as mean centering. Setting this to any non-zero value will set the mean along the feature dimension to be 0. Setting to 0 turns it off. If PCA is performed, data centering is required; if the user selects a PCA calculation but 'center' is off, the function will issue a warning and turn centering on. Options:

- true (default): Centering turned on
- false: Centering turned off

- **scale — Data scaling parameter.** Also known as data normalization. Setting this to a non-zero value to scales each feature to have unit variance prior to PCA. Setting it to 0 turns off data scaling. Options:

- false (default): Scaling turned off
- true: Scaling turned on

For more information on SVM hyperparameters, see Hsu et al. (2003), 'A Practical Guide to Support Vector Classification'. For more info on hyperparamters for random forest, see Matlab documentation for the `treeBagger()` class.²⁴

Outputs

- **M — Classification output struct to be passed into `predict()`.** The struct contains the following subfields:

²⁴<https://www.mathworks.com/help/stats/treebagger.html>

- **classificationInfo**: Additional parameters/info for classification.
 - **mdl**: Classification model which is used in `predict()` to classify labels of new test data.
 - **classifier**: Classifier selected for training.
 - **functionName**: The name of the current function in string format.
 - **cvDataObj**: Object containing data and labels after PCA.
 - **permutations**: Please see 'permutations' section in the input arguments.
 - **ip**: Input parser object for this function.
 - **elapsedTime**: Time elapsed for training current model in seconds. Could be used to gauge permutation testing duration.
 - **maxAccuracy**: Highest classification accuracy obtained during optimization (using `gammaOpt` and `C_opt`).
 - **gammaOpt**: Optimal value for SVM hyperparameter `gamma`.
 - **C_opt**: Optimal value for SVM hyperparameter `C`.
 - **scale**: Please see section for 'scale' input argument.
- **permTestData** — **Permutation testing data to be passed into `predict()`.**

A runnable script with example function calls is provided in the `ExampleFunctionCalls` folder: `example_Classification_trainMulti_opt.m`.

5.10 trainPairs()

This function averages trials in the data matrix `X` on a per-label basis (as defined by elements of `Y`) in groups of `groupSize` trials. In other words, the function averages groups of trials belonging to the same category, where the number of trials averaged in each group is specified by the variable `groupSize`.

- The user can also optionally enter a vector of participant identifiers `P`, in which case trial averaging will additionally be performed on a per-participant basis.
- The function takes in optional name-value pairs to specify handling of remainder trials, whether to shuffle the data after averaging (retaining the mapping between trials and labels), and to set the random number generator.

If the user wishes to shuffle the ordering of data (while preserving labeling of data observations) prior to averaging, they should call the `shuffleData()` function prior to calling this function.

This function uses the helper functions (which can be found in the +Utils folder): `initInputParser()`, `convert2double()`, `verifySVMParameters()`, `subsetTrainTestMatrices()`, `initPairwiseCellMat()`.

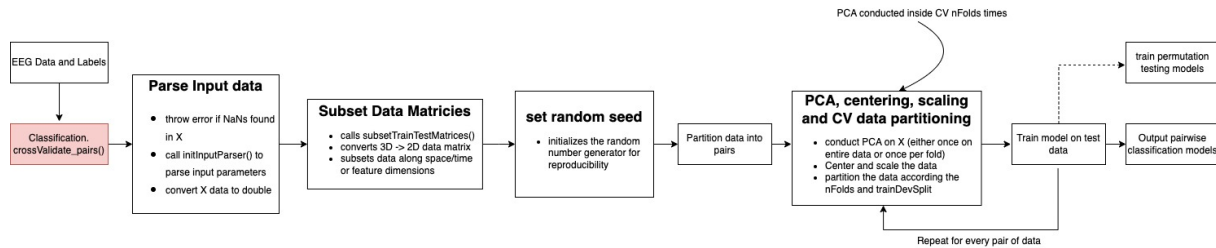


Figure 13: Overview of `trainPairs()` functionality.

Syntax

```
[M, permTestData] = Classification.trainPairs(X, Y, varargin);
```

Required inputs

- **X — M/EEG data matrix.** The X matrix contains the response data. The X matrix can be passed into the function in the following shapes:
 - A 3D sensor-by-time-by-trial matrix.
 - A 2D trial-by-feature matrix. For example, this matrix could contain response activations at a single time sample recorded from multiple sensors, or across time from a single sensor. In the function code, 2D data matrices are effectively treated as trial-by-time (single-sensor) matrices.
- **Y — Labels vector.** The Y vector contains the numeric labels corresponding to each trial in the M/EEG data matrix X. Both row and column vectors will be accepted. The length of Y must correspond to the size of X along the trial dimension.

Optional inputs

- **P — Participant vector.** The optional input P specifies the participant identifier of every trial of data. The length of P must correspond to the length of Y and the length of the trial dimension of X. If P is not entered or is empty, the function will return NaN as randomized P. P can be a numeric vector, string array, or cell array.

Optional name-value inputs

- **timeUse — Temporal subset to classify (3D input).** If X is a 3D, space-by-time-by-trial matrix, then this option will subset X along the time dimension. The input argument should be passed in as a vector of indices that indicate the time dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 2D, trial-by-feature matrix.

- **spaceUse — Spatial subset to classify (3D input).** If X is a 3D, space-by-time-by-trial matrix, then this option will subset X along the space dimension. The input argument should be passed in as a vector of indices that indicate the space dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 2D, trial-by-feature matrix.
- **featureUse — Temporal subset to classify (2D input).** If X is a 2D, trial-by-feature matrix, then this option will subset X along the features dimension. The input argument should be passed in as a vector of indices that indicate the feature dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 3D, space-by-time-by-trial matrix.
- **rngType — Random number generator (rng) specification.** If rngType is not entered or is empty, rng will be assigned here as {'shuffle', 'twister'}. The rngType input can be specified in the following ways:
 - Single acceptable rng specification input (e.g., 1, 'default', 'shuffle'); in these cases, the generator will be set to 'twister'.
 - Dual-argument specifications as either a 2-element cell array (e.g., {'shuffle', 'twister'}) or string array (e.g., ["shuffle", "twister"]).
 - rng struct as previously assigned by rngType = rng.
- **PCA — Principal Component Analysis specification.** Set principal component analysis on data matrix X. To retain components that explain a certain percentage of variance, enter a decimal value [0, 1). To retain a certain number of principal components, enter an integer greater or equal to 1. Default value is .99, which selects principal components that explain 99% of the variance. Enter 0 to disable PCA. PCA is computed along the feature dimension – that is, along the column dimension of the trial-by-feature matrix that is input to the classifier. Options:
 - Decimal between [0, 1) (default value 0.99): Include as many features as explain $N * 100$ percent of the variance of the data.
 - Integer greater than or equal to 1: Include the first N features.
 - 0: Do not perform PCA.
- **PCAIinFold — When PCA is computed.** This input specifies whether PCA is conducted on the training partition of each fold during cross validation, or if PCA is conducted once on the entire dataset prior to partitioning data for cross validation. Options:
 - true: Conduct PCA within each fold.

- `false` (default): Conduct PCA once across the entire data matrix X .
- **nFolds — Number of cross-validation folds.** Must be an integer greater than 1 and less than or equal to the number of trials. Default is 10.
- **classifier — Classifier used during cross validation.** Supported classifiers include support vector machine (SVM), linear discriminant analysis (LDA) and random forest (RF). Options:
 - `'LDA'` (default): Linear discriminant analysis.
 - `'SVM'`: Support vector machine. If using this classifier, the user must manually specify hyperparameter C (linear, rbf kernels) and γ (rbf kernel). If parameters are not known, use the `crossValidateMulti_opt` function to optimize SVM hyperparameters.
 - `'RF'`: Random forest.

Note for SVM classifications: If, when performing any SVM classification, 1) you receive an error message regarding an SVM function not being found, 2) the function hangs (no changes to the pop-up progress bar for several minutes), or 3) classifier performance is unexpectedly poor, you may still need to set up LIBSVM. See Chapter 2.2.3 for more information.

- **kernel — Decision function specification (SVM classifier).** This input is only relevant if the SVM classifier is selected. Options:
 - `'linear'`: Hyperparameter C
 - `'rbf'` (default): Hyperparameters γ (gamma) and C

For more information on the SVM classification kernels, see Hsu, Chang and Lin's 2003 paper, "A Practical Guide to Support Vector Classification".

- **gamma — Hyperparameter of the rbf kernel for SVM classification.** If SVM is selected as the classifier, and rbf is selected as the kernel, then gamma must be manually set by the user. It is recommended for the user to run the corresponding `'_opt'` function to determine a suitable gamma value.
- **C — Hyperparameter rbf, linear kernels for SVM classification.** If SVM is selected as the classifier, then C must be manually set by the user. It is recommended for the user to run the corresponding `'_opt'` function to determine a suitable C value.
- **numTrees — Hyperparameter of the random forest classifier.** This chooses the number of decision trees to grow. Default is 128.

- **minLeafSize** — **Hyperparameter of the random forest classifier.** Choose the minimum number of observations per tree leaf. Default is 1.
- **permutations** — **Number of permutations during permutation testing.** If this value is set to 0, then permutation testing will be turned off. If it is set to an integer n greater than 0, then classification will be performed over n permutation iterations. Default value is 0 (off).

- - - Implementation notes - - -

The permutation testing for this function is carried out by the `predict()` function, consistent with the steps described in the `crossValidatePairs()` function. Please refer to the permutations section in the `crossValidatePairs()` function documentation or the code docstring to learn more.

- **center** — **Data centering parameter.** Also known as mean centering. Setting this to any non-zero value will set the mean along the feature dimension to be 0. Setting to 0 turns it off. If PCA is performed, data centering is required; if the user selects a PCA calculation but 'center' is off, the function will issue a warning and turn centering on. Options:

- `true` (default): Centering turned on
- `false`: Centering turned off

- **scale** — **Data scaling parameter.** Also known as data normalization. Setting this to a non-zero value to scales each feature to have unit variance prior to PCA. Setting it to 0 turns off data scaling. Options:

- `false` (default): Scaling turned off
- `true`: Scaling turned on

Outputs

- **M** — **Pairwise classification output struct to be passed into `predict()`.** The struct contains the following subfields:
 - **classificationInfo**: Additional parameters/info for classification.
 - **mdl**: Classification model which is used in `predict()` to classify labels of new test data.
 - **classifier**: Classifier selected for training.
 - **functionName**: The name of the current function in string format.
 - **cvDataObj**: Object containing data and labels after PCA.

- **permutations**: Please see 'permutations' section in the input arguments.
 - **ip**: Input parser object for this function.
 - **elapsedTime**: Time elapsed for training current model in seconds. Could be used to gauge permutation testing duration.
 - **scale**: Please see section for 'scale' input argument.
- **permTestData** — **Permutation testing data to be passed into predict()**.

A runnable script with example function calls is provided in the ExampleFunctionCalls folder: `example_Classification_trainPairs.m`.

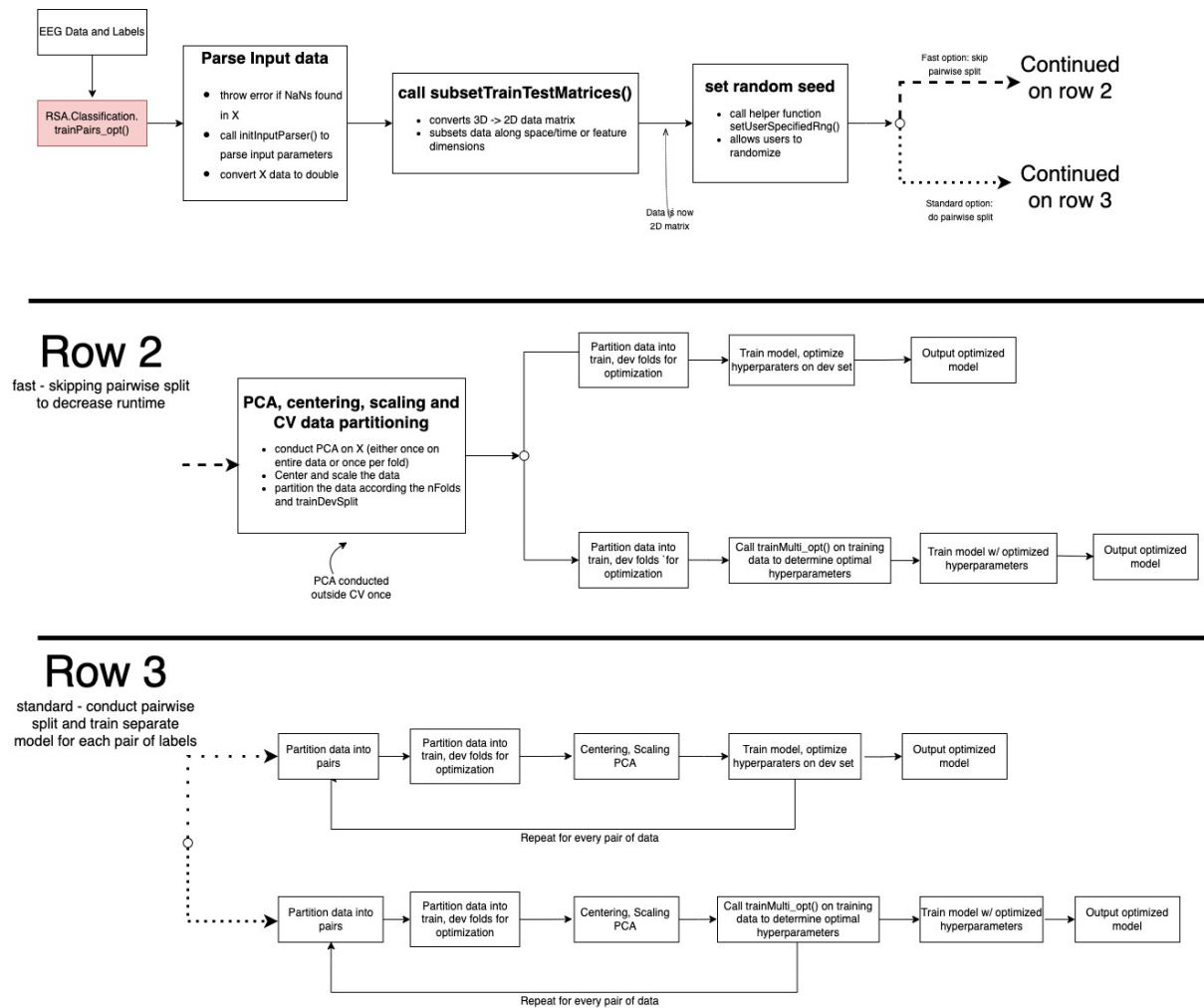
5.11 trainPairs_opt()

This function averages trials in the data matrix X on a per-label basis (as defined by elements of Y) in groups of `groupSize` trials. In other words, the function averages groups of trials belonging to the same category, where the number of trials averaged in each group is specified by the variable `groupSize`.

- The user can also optionally enter a vector of participant identifiers P , in which case trial averaging will additionally be performed on a per-participant basis.
- The function takes in optional name-value pairs to specify handling of remainder trials, whether to shuffle the data after averaging (retaining the mapping between trials and labels), and to set the random number generator.

If the user wishes to shuffle the ordering of data (while preserving labeling of data observations) prior to averaging, they should call the `shuffleData()` function prior to calling this function.

This function uses the helper functions (which can be found in the `+Utils` folder): `initInputParser()`, `convert2double()`, `subsetTrainTestMatrices()`, `initPairwiseCellMat()`.

Figure 14: Overview of `trainPairs_opt()` functionality.

Syntax

```
[M, permTestData] = RSA.Classification.trainPairs_opt(X, Y, varargin);
```

Required inputs

- **X — M/EEG data matrix.** The X matrix contains the response data. The X matrix can be passed into the function in the following shapes:
 - A 3D sensor-by-time-by-trial matrix.
 - A 2D trial-by-feature matrix. For example, this matrix could contain response activations at a single time sample recorded from multiple sensors, or across time from a single sensor. In the function code, 2D data matrices are effectively treated as trial-by-time (single-sensor) matrices.
- **Y — Labels vector.** The Y vector contains the numeric labels corresponding to each trial in the M/EEG data matrix X. Both row and column vectors will be accepted. The length of Y must correspond to the size of X along the trial dimension.

Optional name-value inputs

- **rngType — Random number generator (rng) specification.** If rngType is not entered or is empty, rng will be assigned here as {'shuffle', 'twister'}. The rngType input can be specified in the following ways:
 - Single acceptable rng specification input (e.g., 1, 'default', 'shuffle'); in these cases, the generator will be set to 'twister'.
 - Dual-argument specifications as either a 2-element cell array (e.g., {'shuffle', 'twister'}) or string array (e.g., ["shuffle", "twister"]).
 - rng struct as previously assigned by rngType = rng.
- **PCA — Principal Component Analysis specification.** Set principal component analysis on data matrix X. To retain components that explain a certain percentage of variance, enter a decimal value [0, 1). To retain a certain number of principal components, enter an integer greater or equal to 1. Default value is .99, which selects principal components that explain 99% of the variance. Enter 0 to disable PCA. PCA is computed along the feature dimension – that is, along the column dimension of the trial-by-feature matrix that is input to the classifier. Options:
 - Decimal between [0, 1) (default value 0.99): Include as many features as explain $N * 100$ percent of the variance of the data.
 - Integer greater than or equal to 1: Include the first N features.
 - 0: Do not perform PCA.
- **nFolds — Number of cross-validation folds.** Must be an integer greater than 1 and less than or equal to the number of trials. Default is 10.
- **timeUse — Temporal subset to classify (3D input).** If X is a 3D, space-by-time-by-trial matrix, then this option will subset X along the time dimension. The input argument should be passed in as a vector of indices that indicate the time dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 2D, trial-by-feature matrix.
- **spaceUse — Spatial subset to classify (3D input).** If X is a 3D, space-by-time-by-trial matrix, then this option will subset X along the space dimension. The input argument should be passed in as a vector of indices that indicate the space dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 2D, trial-by-feature matrix.

- **featureUse — Temporal subset to classify (2D input).** If X is a 2D, trial-by-feature matrix, then this option will subset X along the features dimension. The input argument should be passed in as a vector of indices that indicate the feature dimension indices that the user wants to subset. This argument will not do anything if input matrix X is a 3D, space-by-time-by-trial matrix.
- **rngType — Random number generator (rng) specification.** If `rngType` is not entered or is empty, `rng` will be assigned here as `{'shuffle', 'twister'}`. The `rngType` input can be specified in the following ways:
 - Single acceptable `rng` specification input (e.g., `1`, `'default'`, `'shuffle'`); in these cases, the generator will be set to `'twister'`.
 - Dual-argument specifications as either a 2-element cell array (e.g., `{'shuffle', 'twister'}`) or string array (e.g., `["shuffle", "twister"]`).
 - `rng` struct as previously assigned by `rngType = rng`.
- **PCA — Principal Component Analysis specification.** Set principal component analysis on data matrix X . To retain components that explain a certain percentage of variance, enter a decimal value $[0, 1)$. To retain a certain number of principal components, enter an integer greater or equal to 1. Default value is .99, which selects principal components that explain 99% of the variance. Enter 0 to disable PCA. PCA is computed along the feature dimension – that is, along the column dimension of the trial-by-feature matrix that is input to the classifier. Options:
 - Decimal between $[0, 1)$ (default value 0.99): Include as many features as explain $N * 100$ percent of the variance of the data.
 - Integer greater than or equal to 1: Include the first N features.
 - 0: Do not perform PCA.
- **PCAIinFold — When PCA is computed.** This input specifies whether PCA is conducted on the training partition of each fold during cross validation, or if PCA is conducted once on the entire dataset prior to partitioning data for cross validation. Options:
 - `true`: Conduct PCA within each fold.
 - `false` (default): Conduct PCA once across the entire data matrix X .
- **nFolds — Number of cross-validation folds.** Must be an integer greater than 1 and less than or equal to the number of trials. Default is 10.
- **classifier — Classifier used during cross validation.** Currently, only SVM is supported for hyperparameter optimization:

- 'SVM' (default)

Note for SVM classifications: If, when performing any SVM classification, 1) you receive an error message regarding an SVM function not being found, 2) the function hangs (no changes to the pop-up progress bar for several minutes), or 3) classifier performance is unexpectedly poor, you may still need to set up LIBSVM. See Chapter 2.2.3 for more information.

- **kernel — Decision function specification (SVM classifier).** This input is only relevant if the SVM classifier is selected. Options:

- 'linear': Hyperparameter C
- 'rbf' (default): Hyperparameters γ (gamma) and C

For more information on the SVM classification kernels, see Hsu, Chang and Lin's 2003 paper, "A Practical Guide to Support Vector Classification".

- **optimization — Optimization data specification.** This parameter controls whether optimization is conducted using a full *nFolds* cross validation on the training data or using a single development fold. Options:

- 'singleFold' (default)
- 'nestedCV'

- **gammaSpace — Gamma search space.** Vector of *gamma* values to search over during hyperparameter optimization. *Gamma* is a hyperparameter of the rbf kernel for SVM classification. Default is 5 logarithmically spaced points between 10^{-5} and 10^5 .
- **cSpace — C search space.** Vector of C values to search over during hyperparameter optimization. See input argument C is a hyperparameter of both the rbf and linear kernels for SVM classification. Default is 5 logarithmically spaced points between 10^{-5} and 10^5 .
- **permutations — Number of permutations during permutation testing.** If this value is set to 0, then permutation testing will be turned off. If it is set to an integer n greater than 0, then classification will be performed over n permutation iterations. Default value is 0 (off).

- - - Implementation notes - - -

The permutation testing for this function is carried out by the `predict()` function, consistent with the steps described in the `crossValidatePairs_opt()` function. Please refer to the permutations section in the `crossValidatePairs_opt()` function documentation or the code docstring to learn more.

- **center — Data centering parameter.** Also known as mean centering. Setting this to any non-zero value will set the mean along the feature dimension to be 0. Setting to 0 turns it off. If PCA is performed, data centering is required; if the user selects a PCA calculation but 'center' is off, the function will issue a warning and turn centering on. Options:
 - true (default): Centering turned on
 - false: Centering turned off
- **scale — Data scaling parameter.** Also known as data normalization. Setting this to a non-zero value to scales each feature to have unit variance prior to PCA. Setting it to 0 turns off data scaling. Options:
 - false (default): Scaling turned off
 - true: Scaling turned on

Outputs

- **M — Pairwise classification output struct to be passed into predict().** The struct contains the following subfields:
 - **classificationInfo:** Additional parameters/info for classification.
 - **mdl:** Classification model which is used in predict() to classify labels of new test data.
 - **classifier:** Classifier selected for training.
 - **functionName:** The name of the current function in string format.
 - **cvDataObj:** Object containing data and labels after PCA.
 - **permutations:** Please see 'permutations' section in the input arguments.
 - **ip:** Input parser object for this function.
 - **elapsedTime:** Time elapsed for training current model in seconds. Could be used to gauge permutation testing duration.
 - **maxAccuracy:** Highest classification accuracy obtained during optimization (using gammaOpt and C_opt).
 - **gammaOpt:** Optimal value for SVM hyperparameter gamma.
 - **C_opt:** Optimal value for SVM hyperparameter C.
 - **scale:** Please see section for 'scale' input argument.
- **permTestData — Permutation testing data to be passed into predict().**

A runnable script with example function calls is provided in the ExampleFunctionCalls folder: `example_Classification_trainPairs_opt.m`.

5.12 predict()

Given a MatClassRSA classification model and input data X, this function will predict the labels of the trials contained in X.

This function uses the helper functions (which can be found in the +Utils folder): `is2Dor3DMatrix()`, `subsetTrainTestMatrices()`, `setUserSpecifiedRng()`, `centerAndScaleData()`, `modelPredict()`, `computeAccuracy()`, `fitModel()`, `modelPredict()`, `computeAccuracy()`, `permTestPVal()`, `trainDevGridSearch()`, `nestedCvGridSearch()`, `initPairwiseCellMat()`, `decValues2PairwiseAcc()`.

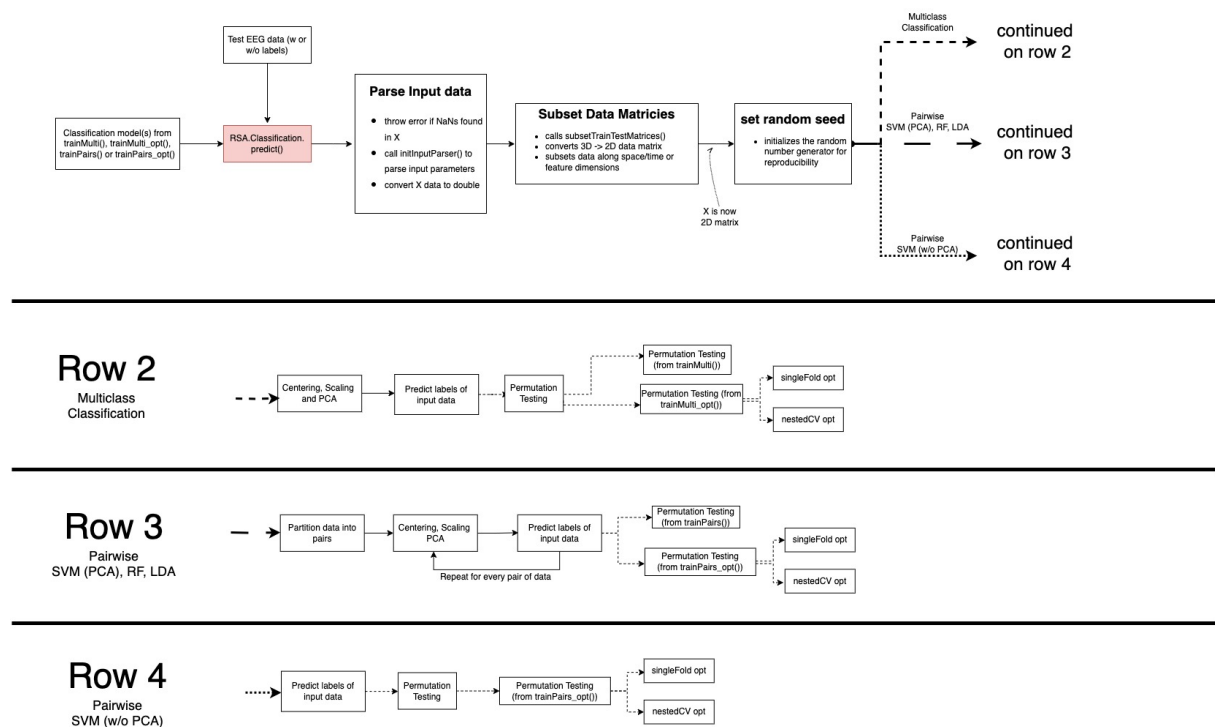


Figure 15: Overview of `predict()` functionality.

Syntax

```
P = predict(M, X, varargin);
```

Required inputs

- **M — Classification model.** MatClassRSA classification model output from `trainMulti()`, `trainMulti_opt()`, `trainPairs()`, or `trainPairs_opt()`.
- **X — M/EEG data matrix.** The X matrix contains the response data. The X matrix can be passed into the function in the following shapes:
 - A 3D sensor-by-time-by-trial matrix.

- A 2D trial-by-feature matrix. For example, this matrix could contain response activations at a single time sample recorded from multiple sensors, or across time from a single sensor. In the function code, 2D data matrices are effectively treated as trial-by-time (single-sensor) matrices.

Optional name-value inputs

- **actualLabels** — **Vector of trial labels of X.** The length of `actualLabels` must match the length of the trial dimension of `X`.
- **permTestData** — **Data used to conduct permutation testing.** This corresponds to the last output from `trainMulti_opt()`, `trainPairs()`, or `trainPairs_opt()`. This input is required if permutation testing is turned on.

- - - Note on permutation testing - - -

Permutation testing in this function depends on which function and classifier was used to train the classification model (i.e., `trainMulti()`, `trainMulti_opt()`, `trainPairs()`, `trainPairs_opt()`). Specifically, for each `train` function, details of permutation testing are given in the corresponding `crossValidate` function. For instance, if `trainMulti()` was used, then the permutation testing implementation is designed to match that of `crossValidateMulti()`. Please refer to the permutations section in the corresponding `crossValidate` function documentation or the code docstring to learn more.

Outputs

- **P** — **Prediction output.** The fields in this struct will vary depending on which function was used to train the model `M`.

If the classification model `M` was created using `trainMulti()` or `trainMulti_opt()`, `P` will contain the following fields:

- **predY:** Predicted labels for the input data
- **accuracy:** Accuracy of predicted values compared to actual labels specified in `actualLabels`
- **CM:** Matrix detailing predicted labels compared to actual labels specified in `actualLabels`
- **predictionInfo:** Contains prediction related information
- **classificationInfo:** Contains classification related information
- **model:** Contains classification model(s)

- **permAccs**: Permutation testing accuracies. This field will be NaN if permutation testing is not specified.
- **classificationInfo**: This struct contains the specifications used during classification, including PCA, PCAinFold, nFolds, classifier, and dataPartitionObj.
- **dataPartitionObj**: This struct contains the train/test data partitions for cross validation (and a dev data partition if hyperparameter optimization is specified).

Note that accuracy and CM can be computed only if the optional input `actualLabels` was provided. If `actualLabels` was not provided, these outputs will be NaN.

If `M` was created using `trainPairs()` or `trainPairs_opt()`, `P` will contain the following fields:

- **AM**: This is a symmetric matrix containing the pairwise accuracies.
- **modelsConcat**: Contains a concatenated list of classification model(s)
- **predictionInfo**: Contains prediction related information
- **classificationInfo**: Contains classification related information
- **pValMat**: contains a matrix of p-values, in which each off-diagonal element corresponds to every pair of labels.
- **permAccs**: Permutation testing accuracies. This field will be NaN if permutation testing is not specified.
- **classificationInfo**: This struct contains the specifications used during classification, including PCA, PCAinFold, nFolds, classifier, and dataPartitionObj.
- **dataPartitionObj**: This struct contains the train/test data partitions for cross validation (and a dev data partition if hyperparameter optimization is specified).

A runnable script with example function calls is provided in the `ExampleFunctionCalls` folder: `example_Classification_predict.m`.

Module: RDM Computation

6.1 Overview

Representational Dissimilarity Matrices (RDMs), being matrices that summarize the absolute or (percentile-)ranked distances between all pairs of exemplars (e.g., stimuli), are key to Representational Similarity Analysis. The MatClassRSA toolbox provides multiple functions for computing RDMs. Due to the high dimensionality of M/EEG data, classification may be especially useful for deriving RDMs, as this approach provides an optimized distance measure by weighting available features to maximize classifier accuracy on the training data. But M/EEG RDMs can also be computed by other means including correlation (Weber et al., 2024; Guggenmos et al., 2018) or by computing distances between ERP voltages (Groen et al., 2012) or spatiotemporal M/EEG feature vectors (Guggenmos et al., 2018).

As shown in the Figure 16 module overview, the `computeCMRDM()` and `shiftPairwiseAccuracyRDM()` functions operate on outputs from the Classification module, specifically the multiclass confusion matrices and pairwise accuracy matrices, respectively. This module also includes functionalities for computing RDMs directly from input M/EEG input data—which could optionally have been passed through one or more Preprocessing functions—with the `computeEuclideanRDM()` and `computePearsonRDM()` functions.

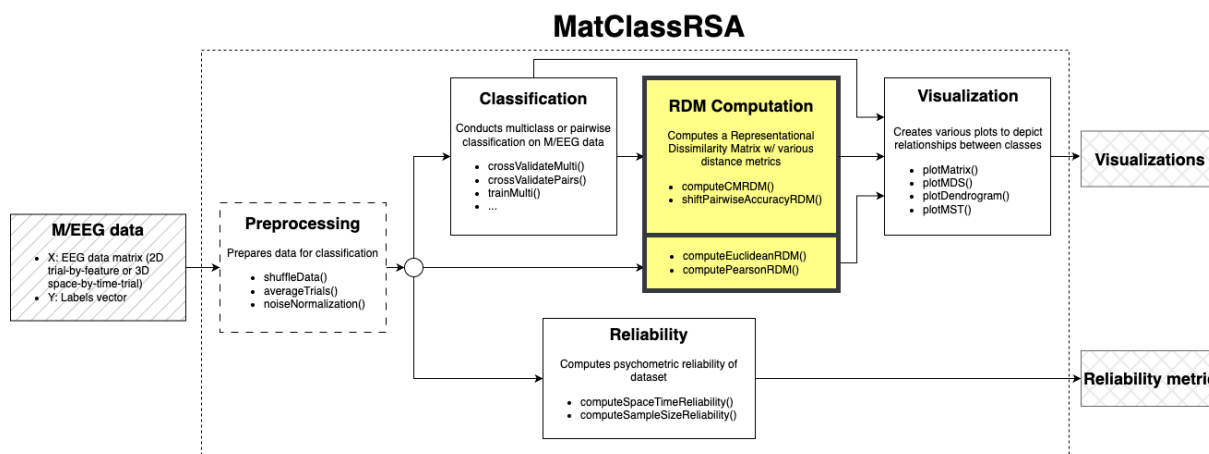


Figure 16: Position of RDM Computation module in MatClassRSA toolbox.

6.2 computeCMRDM()

This function transforms and scales a generalized square proximity matrix. The default specifications assume a square multicategory confusion matrix as input, but the function can be used on other types of square proximity matrices as well (e.g., matrices of pairwise similarity ratings, or un-ranked RDMs).

This function uses the helper functions `convertSimToDist()`, `normalizeMatrix()`, `rankDistances()`, and `symmetrizeMatrix()`.

Syntax

```
[RDM, params] = computeCMRDM(M, varargin);
```

Required inputs

- **M — Square input matrix.** A square matrix, typically thought to be a multicategory confusion matrix but could also be a matrix of e.g., pairwise correlations or distances if using custom name-value input specifications. If inputting a confusion matrix, the matrix should be arranged such that rows represent actual labels and columns represent predicted labels (e.g., element (3, 1) denotes the number of observations actually from class 3 that were predicted to be from class 1) – this is the orientation output by the MatClassRSA classification functions.

Optional name-value pair inputs

- **normalize — Division of matrix entries by a specified value.** Options:

- 'sum' (default): Divide each matrix element by the sum of the respective row, so that each row sums to 1. Assuming that confusion matrices arrange actual labels as rows and predicted labels as columns, this procedure computes the estimated conditional probabilities: $P(\text{predicted}|\text{actual})$ (Shepard, 1958b).

Note: If the sum of any row is zero, the normalization subfunction will print a warning and the outputs of all elements in those rows will be zeros (not NaNs).

- 'diagonal' — Divide each matrix element by the diagonal value in the respective row. For confusion matrices, this produces self-similarity of 1 (Shepard, 1958a).

Note: If any element along the diagonal is zero, this option will introduce NaNs in the output; in this case the normalization subfunction will issue an error and advise the user to use 'sum' normalization instead.

Note: If any off-diagonal element in a matrix row exceeds the diagonal value in that row (which would produce off-diagonal values greater than 1 after normalization), the the function will print a warning and advise the user to use 'sum' normalization instead.

- 'none' — Do not perform any normalization of the matrix. This option may make sense when the input matrix entries are already akin to the outputs of the other normalization options (e.g., a correlation matrix).

- **symmetrize** — **Ensure that the distance between i, j equals the distance between j, i .** There are options to compute the arithmetic, geometric, or harmonic mean of the input matrix and its transpose. Options:

- 'arithmetic' (default): For input matrix M , compute $(M + M^T)/2$.
- 'geometric': For input matrix M , compute $\sqrt{(M .* M^T)}$.
- 'harmonic': For input matrix M , compute $2 * M .* M^T ./ (M + M^T)$.

Note: In this case, any zeros on the diagonal will be converted to NaNs, and the symmetrize subfunction will print a warning.

- 'none': Do not symmetrize the matrix (e.g., if the input is already symmetric).

- **distance** — **Convert similarity matrices to distance matrices.** Similarities are assumed to already be in the range of -1 to 1 (or 0 to 1 for non-negative distances only). Options:

- 'linear' (default): Computes distance $D = 1 - S$.
- 'power': Computes distance $D = 1 - S^{\text{distpower}}$ (see below for specification of distpower).

- 'logarithmic': Computes distance

$$D = \log_2(\text{distpower} * M + 1) / \log_2(\text{distpower} + 1)$$
 (see below for specification of distpower).
- 'none': Do not perform any distance conversion (for example, if values in the put matrix are already distances).
- **distpower — Integer greater than zero.** Distpower is used in the 'power' and 'logarithmic' options of the distance computations (see above). The default value is 1. If the parameter is not input as a positive integer, the function will override it with 1 and issue a warning.
- **rankdistances — Transform RDM values to ranks or percentile ranks.** The distances of an RDM are sometimes transformed to ranks or percentile ranks for visualizing the data. MatClassRSA rank operations assume a symmetric input matrix, and operate on the lower triangle of the input (not including the diagonal). Options:
 - 'none' (default): Do not rank the matrix elements.
 - 'rank': Return the ranked values, adjusted for ties. If the input matrix is not symmetric, the subfunction will issue a warning and operate only on the lower triangle of the matrix, returning a symmetric matrix.
 - 'percentrank': Return the ranked distances, adjusted for ties and divided by the number of unique pairs represented in the matrix (i.e., the number of elements in the lower triangle of the matrix, excluding the diagonal). If the input matrix is not symmetric, the subfunction will issue a warning and operate only on the lower triangle of the matrix, returning a symmetric matrix.

Outputs

- **RDM — The Representational Dissimilarity Matrix (distance matrix).** RDM is a square matrix of the same size as the input matrix M.
- **params — RDM computation parameters.** This is a struct whose fields contain the normalization, symmetrization, distance measure, distance power, and ranking specifications.

Example function calls

Call the function on an input confusion matrix, using all default specifications (sum normalization, arithmetic symmetrize, linear distance, no ranks):

```
[RDM, params] = computeCMRDM(M);
```

Call the function on an input confusion matrix, using default normalization and symmetrize parameters, but specifying power distance with distpower of 2 and percent-rank distances.

```
[RDM, params] = computeCMRDM(M, 'distance', 'power', ...  
'distpower', 2, 'rankdistances', 'percentrank');
```

Call the function on an input correlation matrix. In this case the matrix is already normalized and symmetric. The following function call specifies to skip those two steps, use the default linear distance (omitted from function call), and return ranked distances.

```
[RDM, params] = computeCMRDM(M, 'normalize', 'none', ...  
'symmetrize', 'none', 'rankdistances', 'rank');
```

A runnable script with example function calls is provided in the ExampleFunctionCalls folder: `example_RDM_Computation_computeCMRDM.m`.

6.3 shiftPairwiseAccuracyRDM()

Given an input matrix of pairwise accuracies, this function subtracts 0.5 from the off-diagonal entries.

This function does not use any helper functions.

Syntax

```
xShift = shiftPairwiseAccuracyRDM(xIn, varargin);
```

Required inputs

- **xIn** — **Square input matrix.** The input is generally assumed to be a matrix of pairwise classification accuracies. Off-diagonal values $< i, j >$ for $i \neq j$ represent pairwise classification accuracies between categories i and j . Values on the diagonal are assumed to be NaN or zero. The input matrix need not be symmetric.

Optional name-value inputs

- **pairScale** — **Specification of input data range.** The user can set this value to 1 if the input data are on a 0-to-1 scale (where 0.5 would denote chance), or set to 100 if the input data are on a 0-to-100 scale (where 50.0 would denote chance). Note that if any value of the input matrix is found to be greater than 1, pairScale will be set to 100 during the function run, potentially overriding the user or default specification.
 - 1 (default): Input matrix values are scaled between 0 and 1.
 - 100: Input matrix values are scaled between 0 and 100.

Outputs

- **xShift** — **Shifted-value RDM**. This is a matrix the same size as the input `xIn`. Off-diagonal values are input values minus 0.5. Entries on the diagonal will be NaN for any entries that were NaN in the input matrix diagonal; otherwise, diagonal entries of `xShift` will be zero, whether or not the input diagonal entries were zero.

Example function calls

The following function call will use default specifications of the function (i.e., input data are scaled between 0 and 1):

```
RDM = RDM_Computation.shiftPairwiseAccuracyRDM(pairAcc_1);
```

The following function call specifies `pairScale` for input data scaled between 0 and 100:

```
RDM = RDM_Computation.shiftPairwiseAccuracyRDM(pairAcc_100, ...
    'pairScale', 100);
```

A runnable script with example function calls is provided in the `ExampleFunctionCalls` folder: `example_RDM_Computation_shiftPairwiseAccuracyRDM.m`.

6.4 computeEuclideanRDM()

This function returns pairwise similarities with respect to cross-validated Euclidean distance. For example, the output of this function provides an estimate of the Euclidean distance between the two pattern vectors for each of two stimuli. In this function, we adapt the ‘cross-validated’ method for computing Euclidean distance (Guggenmos et al., 2018). In each permutation, the trials are randomly split into two partitions – one partition is the ‘train’ partition and the other is the ‘test’ partition. More concretely, if a single image is repeated $nTrials$ times, the train (and test) partitions would each be a matrix of dimensions $nTrials/2 * nElectrodes$, each of which would then be subsequently averaged across the trials dimension, resulting in two vectors of length $nElectrodes$ (one ‘train’ vector and one ‘test’ vector). To obtain the cross-validated Euclidean distance between a pair of pattern vectors (e.g., responses to a pair of images), the difference between the pattern vectors for the two conditions in their corresponding partition is first computed. That is, the difference between the ‘train’ (‘test’) vector for image A and the ‘train’ (‘test’) vector for image B is first computed. The cross-validated Euclidean distance is then the inner product between the two difference vectors. This method improves distance estimates since it removes noise components via this cross-validation scheme (Guggenmos et al., 2018).

A possible data input to this function would have dimensions $nElectrodes \times (nTrials * nClasses)$. With this input, the resulting RDM would be computed using the electrode values at a specific time point as features. On the other hand, one could also provide, as input, data of dimensions $nTimepoints \times (nTrials * nClasses)$. In this case, the resulting RDM would be computed using the time point values for a particular electrode as features.

This function is adapted from code provided by Guggenmos et al. (2018).²⁵ This function uses the helper function `setUserSpecifiedRng()`.

Syntax

```
D = computeEuclideanRDM(X, Y, varargin);
```

Required inputs

- **X — Input data matrix.** The size of X should be $nFeatures \times nTrials$. Users working with 3D data matrices should already have subset the data along a single sensor (along the space dimension) or a time sample (along the time dimension).
- **Y — Labels vector.** The Y vector contains the numeric labels corresponding to each trial in the M/EEG data matrix X. Both row and column vectors will be accepted. The length of Y must correspond to the size of X along the trial dimension.

Optional name-value inputs

- **nPermutations — number of random train-test permutations.** If not entered or empty, will default to 10.
- **rngType — Random number generator (rng) specification.** If `rngType` is not entered or is empty, `rng` will be assigned here as `{'shuffle', 'twister'}`. The `rngType` input can be specified in the following ways:
 - Single acceptable `rng` specification input (e.g., `1`, `'default'`, `'shuffle'`); in these cases, the generator will be set to `'twister'`.
 - Dual-argument specifications as either a 2-element cell array (e.g., `{'shuffle', 'twister'}`) or string array (e.g., `["shuffle", "twister"]`).
 - `rng` struct as previously assigned by `rngType = rng`.

Outputs

²⁵https://github.com/m-guggenmos/megmvpa/blob/master/tutorial_matlab/matlab_distance.ipynb

- **D** — output struct containing the average Euclidean RDM across permutations, as well as the per-permutation Euclidean RDMs.
 - **RDM** — the average Euclidean RDM across all user-specified permutations. This $nClasses \times nClasses$ matrix is what a user would typically report as the output RDM.
 - **dissimilarities** — Euclidean distance dissimilarity matrix. Dimensions of this output will be $nClasses \times nClasses \times nPermutations$.

Example function calls

If the user would like to compute reliability across a single electrode or single time point of multi-trial data using default specifications, the function can be called as follows:

```
D = RDM_Computation.computeEuclideanRDM(singleElectrodeX, Y);
D = RDM_Computation.computeEuclideanRDM(singleTimePointX, Y);
```

The user can customize the rng specification (to 3, in this case) and use the default specification for numPermutations as follows:

```
D = RDM_Computation.computeEuclideanRDM(singleElectrodeX, Y, ...
    'rngType', 3);
```

Or, the user can use the default rng specification and customize numPermutations (to 50, in this case):

```
D = RDM_Computation.computeEuclideanRDM(singleElectrodeX, Y, ...
    'numPermutations', 50);
```

Finally, the user can specify both name-value pair inputs:

```
D = RDM_Computation.computeEuclideanRDM(singleElectrodeX, Y, ...
    'rngType', 3, 'numPermutations', 50);
```

A runnable script with example function calls is provided in the ExampleFunctionCalls folder: `example_RDM_Computation_computeEuclideanRDM.m`.

6.5 computePearsonRDM()

This function returns pairwise similarities with respect to cross-validated Pearson correlation. Similar to the procedure for computing the cross-validated Euclidean distance metric, the trials are first randomly split into two partitions (i.e., train and test partitions) in each permutation. The variance and covariance across features (e.g., electrodes) are

computed across partitions and used to compute the cross-validated Pearson's correlation (see Equation 7 in Guggenmos et al., 2018).

A possible data input to this function would have dimensions $nElectrodes \times (nTrials * nClasses)$. With this input, the resulting RDM would be computed using the electrode values at a specific time point as features. On the other hand, one could also provide, as input, data of dimensions $nTimepoints \times (nTrials * nClasses)$. In this case, the resulting RDM would be computed using the time point values for a particular electrode as features.

This function is adapted from code provided by Guggenmos et al. (2018).²⁶ This function uses the helper function `setUserSpecifiedRng()`.

Syntax

```
D = computePearsonRDM(X, Y, varargin);
```

Required inputs

- **X — Input data matrix.** The size of X should be $nFeatures \times nTrials$. Users working with 3D data matrices should already have subset the data along a single sensor (along the space dimension) or a time sample (along the time dimension).
- **Y — Labels vector.** The Y vector contains the numeric labels corresponding to each trial in the M/EEG data matrix X. Both row and column vectors will be accepted. The length of Y must correspond to the size of X along the trial dimension.

Optional name-value inputs

- **nPermutations — number of random train-test permutations.** If not entered or empty, will default to 10.
- **rngType — Random number generator (rng) specification.** If `rngType` is not entered or is empty, `rng` will be assigned here as `{'shuffle', 'twister'}`. The `rngType` input can be specified in the following ways:
 - Single acceptable `rng` specification input (e.g., `1`, `'default'`, `'shuffle'`); in these cases, the generator will be set to `'twister'`.
 - Dual-argument specifications as either a 2-element cell array (e.g., `{'shuffle', 'twister'}`) or string array (e.g., `["shuffle", "twister"]`).
 - `rng` struct as previously assigned by `rngType = rng`.

²⁶https://github.com/m-guggenmos/megmvpa/blob/master/tutorial_matlab/matlab_distance.ipynb

Outputs

- **D** — output struct containing the average Pearson RDM across permutations, as well as the per-permutation Pearson RDMs.
 - **RDM** — the average Pearson RDM across all user-specified permutations. This $nClasses \times nClasses$ matrix is what a user would typically report as the output RDM.
 - **dissimilarities** — Pearson distance dissimilarity matrix. Dimensions of this output will be $nClasses \times nClasses \times nPermutations$.

Example function calls

If the user would like to compute reliability across a single electrode or single time point of multi-trial data using default specifications, the function can be called as follows:

```
D = RDM_Computation.computePearsonRDM(singleElectrodeX, Y);
D = RDM_Computation.computePearsonRDM(singleTimePointX, Y);
```

The user can customize the rng specification (to 3, in this case) and use the default specification for numPermutations as follows:

```
D = RDM_Computation.computePearsonRDM(singleElectrodeX, Y, ...
    'rngType', 3);
```

Or, the user can use the default rng specification and customize numPermutations (to 50, in this case):

```
D = RDM_Computation.computePearsonRDM(singleElectrodeX, Y, ...
    'numPermutations', 50);
```

Finally, the user can specify both name-value pair inputs:

```
D = RDM_Computation.computePearsonRDM(singleElectrodeX, Y, ...
    'rngType', 3, 'numPermutations', 50);
```

A runnable script with example function calls is provided in the ExampleFunctionCalls folder: `example_RDM_Computation_computePearsonRDM.m`.

Module: Visualization

7.1 Overview

In RSA or RSA-like studies, M/EEG classification results are frequently visualized as RDMs derived from all pairwise accuracies or from multiclass confusion matrices. Some studies additionally visualize proximity spaces using multidimensional scaling (MDS), which depicts distances between stimuli in a low-dimensional distance space (Carlson et al., 2013; Cichy et al., 2014; Kaneshiro et al., 2015b), or by means of tree visualizations such as dendrograms, which display the hierarchical structure of the stimuli (Kiani et al., 2007; Kaneshiro et al., 2015b).

The final module of MatClassRSA is the Visualization module, which provides four functions to visualize RDMs. As shown in Figure 17, these functions can operate on outputs of the Classification and RDM Computation modules. Specifically, multiclass confusion matrices and pairwise accuracy matrices output by the Classification module can be input to the `plotMatrix()` visualization function, while RDMs output by the RDM Computation module can be input to all four functions of the Visualization module.

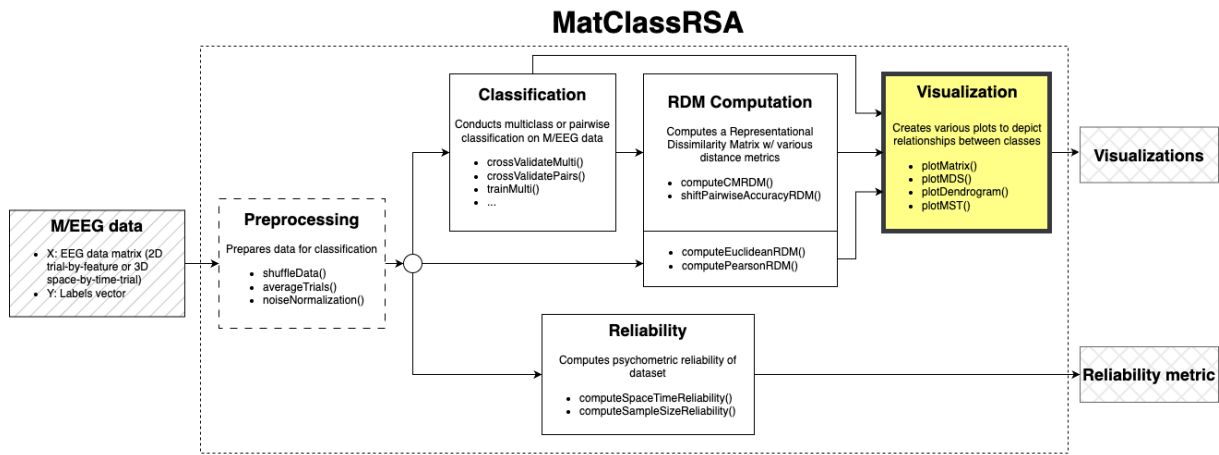


Figure 17: Position of Visualization module in MatClassRSA toolbox.

7.2 Example Figure Code

The function specifications in this chapter include example figures, which are based on the data from `S06.mat` from the example dataset (B. C. Wang et al., 2025a).

The visualized data first underwent data shuffling, noise normalization, and averaging of 30-trial groups:

```
load('S06.mat');
[xShuf, yShuf] = Preprocessing.shuffleData(X, labels6, 'rngType', 7);
xNorm = Preprocessing.noiseNormalization(xShuf, yShuf);
[xAvg, yAvg] = Preprocessing.averageTrials(xNorm, yShuf, 30, ...
    'handleRemainder', 'newGroup', 'rngType', 7);
```

Next, the data were classified in a 6-class classification using LDA, (default) 10-fold cross validation with the PCA parameter set to 0.99 and user-specified rng:

```
M = Classification.crossValidateMulti(xAvg, yAvg, 'PCA', .99, ...
    'classifier', 'LDA', 'rngType', 7);
```

The multiclass confusion matrix was then converted to an RDM using the default specifications of the `computeCMRDM()` function:

```
RDM = RDM_Computation.computeCMRDM(M.CM);
```

Finally, the visualizations use a specified color palette:

```
rgb6 = {[0.1216    0.4667    0.7059], ... % Blue
        [1.0000    0.4980    0.0549] , ... % Orange
```

```
[0.1725    0.6275    0.1725] ,      ... % Green
[0.8392    0.1529    0.1569] ,      ... % Red
[0.5804    0.4039    0.7412] ,      ... % Purple
[0.7373    0.7412    0.1333]};      % Chartreuse
```

The specific function calls for the subsequent visualizations are provided in the function specifications below.

7.3 plotMatrix()

This function creates an image of an input matrix. The matrix can be, for example, a Representational Dissimilarity Matrix (RDM) computed by the functions in the RDM_Computation module, or a confusion matrix or pairwise accuracy matrix output from the functions in the Classification module. An example visualization from this function is provided in Figure 18.

This function uses the helper functions `getTickCoord()` and `rankDistances()`.

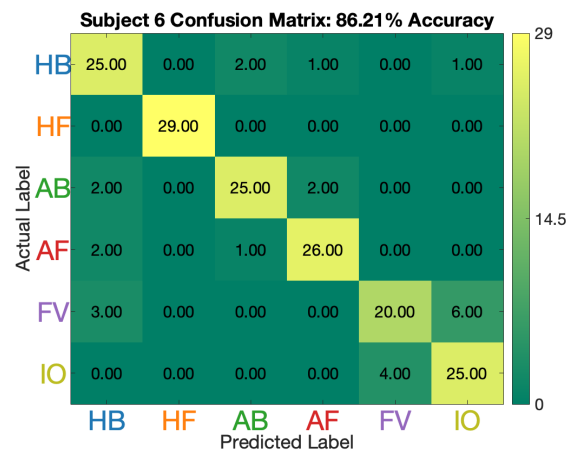


Figure 18: **Example plotMatrix() visualization.** Data from S06.mat underwent 6-class classification after data shuffling, noise normalization, and 30-trial averaging. This matrix image is the multiclass confusion matrix directly output from the classification function, prior to being converted to an RDM. This confusion reflects a high-accuracy classification as most predictions are along the matrix diagonal (actual label equals predicted label). The most off-diagonal entries are in the lower right of the confusion matrix, indicating that misclassifications occurred mainly between the FV and IO classes.

Syntax

```
[img, fig] = plotMatrix(RDM, varargin);
```

Required inputs

- **RDM — Input matrix.** This can be e.g., a confusion matrix or RDM/distance matrix.

Optional name-value inputs

- **ranktype — Whether to convert matrix values to percentile ranks or ranks prior to plotting.** Such conversions are common when visualizing RDMs. Note that conversion of values to ranks or percentile ranks assumes a symmetric input matrix and is based only on values in the lower triangle of the matrix, not including the diagonal. If a non-symmetric matrix is input, a warning is printed and conversion proceeds using only lower-triangle values, returning a symmetric matrix. Options:
 - 'none' (default): Do not rank values of the input matrix.
 - 'rank': Convert matrix values to ranks.
 - 'percentrank': Convert matrix values to percentile ranks.
- **axisColors — Color specifications for row/column labels.** A vector of colors, ordered by the order of labels in the confusion matrix. If this argument is passed in, then square color blocks will be used as the row/column labels. Colors can be expressed as an RGB triplet, short name or long name, e.g. {'y', 'm', 'c', 'r'} or {'yellow', 'magenta', 'cyan', 'red'} or {[1 1 0], [1 0 1], [0 1 1], [1 0 0]}. See Matlab color specification documentation for more information.²⁷
- **axisLabels — Category labels.** A matrix of alphanumeric labels, ordered by same order of items in the confusion matrix e.g., ['cat', 'dog', 'fish'].
- **colorBar — Whether to display colorbar or not.** Default is to not display a colorbar.
 - 0 (default): Hide colorbar
 - 1: Show colorbar
- **matrixLabels — Whether to print values in each matrix element.** If turned on, the value of each matrix entry will be displayed in the matrix. Default is off; enter any value to turn on.
- **FontSize — Font size of matrix and axis labels.** Default value is 15.
- **ticks — Number of ticks in the colorbar, if shown.** Default value is 5.
- **textRotation — Specify rotation angle of text, in degrees.** Default 0 (no rotation).
- **colorBlockSize — Specification of the size of each color block icon.** Default value is 5.

²⁷<https://www.mathworks.com/help/matlab/ref/colormap.html>

- **colorMap — Specify colormap.** This parameter can be used to call a default Matlab colormap, or one specified by the user, to change the overall look of the plot. For example, `plotMatrix(RDM, 'colorMap', 'hsv')`.

Outputs

- **img — Handle of the plot (image) axis.**
- **fig — Handle of the output figure.**

Example function calls

The function call to produce the example given in Figure 18 from confusion matrix `M.CM` is as follows:

```
Visualization.plotMatrix(M.CM, 'colorbar', 1, 'matrixLabels', 1, ...
    'axisLabels', catLabels, 'axisColors', rgb6, ...
    'ticks', 10, 'colorMap', 'summer')
title(sprintf('Subject 6 Confusion Matrix: %.2f%% Accuracy', ...
    M.accuracy*100));
set(gca, 'fontsize', 16)
```

The basic function call with only the required input would be as follows:

```
[img, fig] = Visualization.plotMatrix(CM);
```

This next function call adds a colorbar; axis labels as specified here in the variable `catLabels`; and axis colors as specified here in the variable `rgb6`:

```
[img, fig] = Visualization.plotMatrix(CM, 'colorBar', 1, ...
    'axisLabels', catLabels, 'axisColors', rgb6);
```

The previous function call can be customized further, for instance to add matrix labels and percentile-rank elements of the input matrix:

```
[img, fig] = Visualization.plotMatrix(CM, 'colorBar', 1, ...
    'axisLabels', catLabels, 'axisColors', rgb6, 'ranktype', 'p', ...
    'matrixLabels', 1);
```

The user can continue to add or omit specifications to their preference, for example:

```
[img, fig] = Visualization.plotMatrix(CM, 'colorBar', 1, ...
    'axisLabels', catLabels, 'axisColors', rgb6, 'colorMap', 'summer', ...
    'ticks', 10, 'textRotation', 20);
```

A runnable script with example function calls is provided in the `ExampleFunctionCalls` folder: `example_Visualization_plotMatrix.m`.

7.4 plotMDS()

This function takes in a distance matrix and creates a multidimensional scaling (MDS) plot, which enables the set of pairwise distances represented in the input matrix to be visualized in a low-dimensional representation, where the dimensions are sorted in descending order of importance (Torgerson, 1952; Shepard, 1962). An example visualization from this function is provided in Figure 19.

This function uses the helper function `processRDM()`.

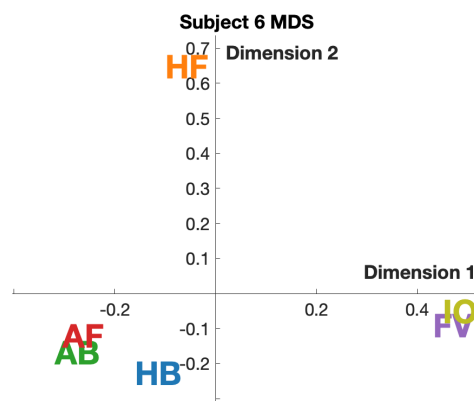


Figure 19: **Example plotMDS() visualization.** The confusion matrix from `S06.mat` 6-class classification was converted to an RDM using default specifications of the `computeCMRDM()` function. This MDS plot includes axis and label customizations as specified in the example function calls below. This visualization plots the coordinates of the stimuli along the 1st and 2nd MDS dimensions. Dimension 1 (x-axis) is primarily separating the two inanimate FV and IO categories from the other, animate categories. It is also separating the two animal categories (AF and AB) from the two human categories (HF and FB). Dimension 2 (y-axis) is mainly separating responses to human faces (HF) from all other categories.

Syntax

```
fig = plotMDS(RDM, varargin);
```

Required inputs

- **RDM — Input distance matrix.** Entries along the diagonal must be 0, and the matrix must be symmetric.

Optional name-value inputs

- **nodeColors — Category color specification.** This input is a vector of colors, whose order corresponds to the order of labels in the confusion matrix. For example, if user

inputs: ['yellow' 'magenta' 'cyan' 'red' 'green' 'blue' 'white' 'black'], then class 1 would be yellow, class 2 would be magenta, and so on. Colors can be expressed as an RGB triplet ([1 1 0]), short name ('y') or long name ('yellow'). See Matlab color specification documentation for more information.²⁸

- **nodeLabels — Category label specification.** A vector of alphanumeric labels, whose order corresponds to the labels in the confusion matrix, e.g., ['cat', 'dog', 'fish'].
- **dimensions — Which MDS dimensions to display.** This input is expected to be a two-element vector specifying which MDS dimensions will be visualized along the x- and y-axes, respectively. Default: [1 2], which represents the two most important dimensions of the MDS.
- **xLim — X-axis limits.** Set the range of the x-axis with a 2-element array, [xMin xMax]. If not entered, the plot will include MATLAB's default x-axis limits, based on the range of values plotted.
- **yLim — Y-axis limits.** Set the range of the y-axis with a 2-element array, [yMin yMax]. If not entered, the plot will include MATLAB's default y-axis limits, based on the range of values plotted.
- **classical — mdscaling specification.** Whether to apply classical and non-classical MDS scaling (mdscaling). View the Matlab documentation for more information.²⁹
 - 1 (default): Apply classical MDS scaling.
 - 0: Apply non-classical MDS scaling.

Outputs

- **fig — Handle of the output figure.**

Example function calls

The function call to produce the example given in Figure 19 from RDM RDM is as follows:

```
Visualization.plotMDS(RDM, ...
    'nodeLabels', catLabels, 'nodeColors', rgb6, 'dimensions', [1 2])
title(sprintf('Subject 6 MDS'))
set(gca, 'fontsize', 16)
```

The basic function call with only the required input would be as follows:

²⁸See footnote 27.

²⁹<https://www.mathworks.com/help/stats/cmdscale.html>

```
fig = Visualization.plotMDS(RDM);
```

Similar to the `plotMatrix()` examples, the user can customize the function call with labels (this time node labels) as specified here in the variable `catLabels` and colors (this time node colors) as specified here in the variable `rgb6`:

```
fig = Visualization.plotMDS(RDM, 'nodeLabels', catLabels, ...  
    'nodeColors', rgb6);
```

The user can also customize which dimensions of the MDS to render in the plot. This example renders dimensions 1 and 3 instead of the default dimensions of 1 and 2:

```
fig = Visualization.plotMDS(RDM, 'nodeLabels', catLabels, ...  
    'nodeColors', rgb6, 'dimensions', [1 3]);
```

The user can specify to use non-classical MDS scaling:

```
fig = Visualization.plotMDS(RDM, 'nodeLabels', catLabels, ...  
    'nodeColors', rgb6, 'classical', 0);
```

The user can also customize the x- and y-axis limits, for instance to impose consistent axes across multiple plots:

```
fig = Visualization.plotMDS(RDM, 'nodeLabels', catLabels, ...  
    'nodeColors', rgb6, 'xLim', [-0.2 1], 'yLim', [-0.5 0.8]);
```

A runnable script with example function calls is provided in the `ExampleFunctionCalls` folder: `example_Visualization_plotMDS.m`.

7.5 plotDendrogram()

Given a distance matrix as input, this function plots a dendrogram, which visualizes the distance structure of the stimuli in a hierarchical fashion (Dubes and Jain, 1980). Each stimulus is shown as a ‘leaf’ of the dendrogram’s tree structure, and the distance between two stimuli can be read as the height one must traverse up the tree in order to get from one leaf to the other. Like MDS plots, dendrograms are useful for displaying how stimuli cluster based on their proximities; their hierarchical structure can also provide insight into distinctions between higher-level groupings (such as the ‘top’ cut in a tree) versus sub-clusters. An example visualization from this function is provided in Figure 20.

This function uses the helper functions `getTickCoord()` and `processRDM()`.

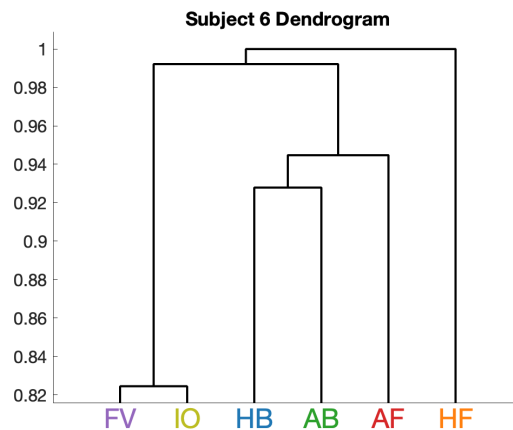


Figure 20: **Example `plotDendrogram()` visualization.** Data from `S06.mat` underwent 6-class classification after data shuffling, noise normalization, and 30-trial averaging. This dendrogram includes normalization (all distances scaled to a maximum value of 1), `ylim` customization, and label and color customizations as specified in the example function calls below. The distance between any two categories is the distance that must be traveled up the tree to go from one category to the other. Interpretation from the leaves up (least to greatest distance): FV and IO categories are most-similar grouping with a distance of around 0.83, followed by HB and AB with a distance of around 0.93. HF is most distant from all other categories, as one must travel all the way up the tree (to a normalized distance of 1) to get to HF from any other category. Interpretation from the top down (greatest to least distance): The top cut in the dendrogram imposes two main category clusters, one comprising HF and the other comprising all other categories. After this, the next cut separates FV and IO from HB, AB, and AF categories, the latter of which further separates HB and AB from AF in a third cut.

Syntax

```
fig = plotDendrogram(RDM, varargin);
```

Required inputs

- **RDM — Input distance matrix.** Entries along the diagonal must be 0, and the matrix must be symmetric.

Optional name-value inputs

- **nodeColors — Category color specification.** This input is a vector of colors, whose order corresponds to the order of labels in the confusion matrix. For example, if user inputs: ['yellow' 'magenta' 'cyan' 'red' 'green' 'blue' 'white' 'black'], then class 1 would be yellow, class 2 would be magenta, and so on. Colors can be expressed as an RGB triplet ([1 1 0]), short name ('y') or long name ('yellow'). See Matlab color specification documentation for more information.³⁰
- **nodeLabels — Category label specification.** A vector of alphanumeric labels, whose order corresponds to the labels in the confusion matrix, e.g., ['cat', 'dog', 'fish'].
- **fontSize — Font size of the labels.** Numeric value. Default is 25.
- **reorder — Specify order of classes.** Must be passed in as a length-N vector, N being the number of classes in RDM. Also, vector should contain values 1:N. Note that custom orderings may disrupt the structure of the dendrogram.
- **yLim — Y-axis limits.** Set the range of the y-axis with a 2-element array, [yMin yMax]. If not entered, the plot will include MATLAB's default y-axis limits, based on the range of values plotted.
- **textRotation — Specify rotation angle of text, in degrees.** Default 0 (no rotation).
- **lineWidth — Dendrogram line width.** Use this parameter to set the width of the lines in the dendrogram. Default 2.
- **lineColor — Dendrogram line color.** Use this parameter to set the color of the lines in the dendrogram. Similar to nodeColors, we can pass in either color abbreviations, full-length color names, or RGB color triplets. Default 'black'.
- **normalization — Scaling the range of distances.** If set to true, this parameter will scale the branch heights (distances) by the maximum distance in the set, such that the maximum displayed distance becomes 1. Default false.

³⁰See footnote 27.

Outputs

- **fig** — **Handle of the output figure.**

Example function calls

The function call to produce the example given in Figure 20 from RDM RDM is as follows:

```
Visualization.plotDendrogram(RDM, ...  
    'nodeLabels', catLabels, 'nodeColors', rgb6)  
title(sprintf('Subject 6 Dendrogram'));
```

The basic function call with only the required input would be as follows:

```
fig = Visualization.plotDendrogram(RDM);
```

As with the `plotMDS()` examples, the user can customize the function call with node labels as specified here in the variable `catLabels` and node colors as specified here in the variable `rgb6`:

```
fig = Visualization.plotDendrogram(RDM, 'nodeLabels', catLabels, ...  
    'nodeColors', rgb6);
```

In the next example function call, the font size is specified to be 25, and the user specifies a custom ordering of nodes as contained here in the vector `order`:

```
fig = Visualization.plotDendrogram(RDM, 'nodeLabels', catLabels, ...  
    'nodeColors', rgb6, 'fontSize', 25, 'reorder', order);
```

Next, normalization is turned on, scaling all the distances to a maximum of 1:

```
fig = Visualization.plotDendrogram(RDM, 'nodeLabels', catLabels, ...  
    'nodeColors', rgb6, 'fontSize', 25, 'normalization', true);
```

Finally, the user can customize other attributes of the text, y-axis limits, and dendrogram lines:

```
fig = Visualization.plotDendrogram(RDM, 'nodeLabels', catLabels, ...  
    'nodeColors', rgb6, 'textRotation', 20, 'yLim', [0.8 1], ...  
    'lineWidth', 2.5, 'lineColor', rgb6{1});
```

A runnable script with example function calls is provided in the `ExampleFunctionCalls` folder: `example_Visualization_plotDendrogram.m`.

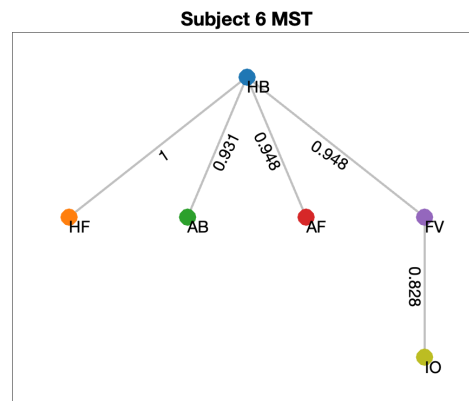


Figure 21: **Example plotMST() visualization.** Data from S06.mat underwent 6-class classification after data shuffling, noise normalization, and 30-trial averaging. This MST includes label and color customizations as specified in the example function calls below. From this MST we can see one more representation of the smallest distance (highest similarity) between the FV and IO categories. Following this, the stimulus categories are more efficiently linked based on their distance to the HB category, including the most-distant category HF.

7.6 plotMST()

Given an RDM input, this function plots a Minimum Spanning Tree (MST). When the set of pairwise distances amongst the input stimuli are represented as a graph, the MST presents the set of vertices that connect all of the stimuli such that the overall distance across the connections is minimized (Gower and Ross, 1969; Cheriton and Tarjan, 1976). An example visualization from this function is provided in Figure 21.

This function uses the helper function `processRDM()`.

Syntax

```
fig = plotMST(RDM, varargin)
```

Required inputs

- **RDM — Input distance matrix.** Entries along the diagonal must be 0, and the matrix must be symmetric.

Optional name-value inputs

- **nodeColors — Category color specification.** This input is a vector of colors, whose order corresponds to the order of labels in the confusion matrix. For example, if user inputs: `['yellow' 'magenta' 'cyan' 'red' 'green' 'blue' 'white' 'black']`, then class 1 would be yellow, class 2 would be magenta, and so on. Colors can be expressed

as an RGB triplet ([1 1 0]), short name ('y') or long name ('yellow'). See Matlab color specification documentation for more information.³¹

- **nodeLabels — Category label specification.** A vector of alphanumeric labels, whose order corresponds to the labels in the confusion matrix, e.g., ['cat', 'dog', 'fish'].
- **edgeLabelSize:** Specify the size of the MST edge labels. Default is 15.
- **nodeLabelSize:** Specify the size of the node labels. Default is 15.
- **nodeLabelRotation:** Set the angle of the node labels. Default is 0.
- **lineWidth — MST tree line width.** Use this parameter to set the width of the lines in the tree. Default 2.
- **lineColor — MST tree line color.** Use this parameter to set the color of the lines in the tree. Similar to nodeColors, the MST tree line color can be expressed as a color abbreviation, full-length color name, or RGB color triplet. Default is [0.5 0.5 0.5].

Outputs

- **fig — Handle of the output figure.**

Example function calls

The function call to produce the example given in Figure 21 from RDM RDM is as follows:

```
Visualization.plotMST(RDM, ...
    'nodeLabels', catLabels, 'nodeColors', rgb6)
title(sprintf('Subject 6 MST'));
```

The basic function call with only the required input would be as follows:

```
fig = Visualization.plotMST(RDM);
```

As with previous Visualization examples, the user can customize the function call with node labels as specified here in the variable catLabels and node colors as specified here in the variable rgb6:

```
fig = Visualization.plotMST(RDM, 'nodeLabels', catLabels, ...
    'nodeColors', rgb6);
```

In the next example function call, the edge label size and node label size are also customized:

³¹See footnote 27.

```
fig = Visualization.plotMST(RDM, 'nodeLabels', catLabels, ...  
    'nodeColors', rgb6, 'edgeLabelSize', 10, 'nodeLabelSize', 25);
```

The user can also customize the line color and width of the MST tree:

```
fig = Visualization.plotMST(RDM, 'nodeLabels', catLabels, ...  
    'nodeColors', rgb6, 'lineWidth', 2.5, 'lineColor', rgb6{1});
```

A runnable script with example function calls is provided in the ExampleFunctionCalls folder: `example_Visualization_plotMST.m`.

Illustrative Examples

To provide further context on how MatClassRSA functions can be used, the toolbox also includes a number of illustrative analyses. These illustrative analyses complement the self-contained MatClassRSA functions documented in previous chapters, showing how the toolbox functions can be combined — and extended — for more comprehensive analyses. This chapter walks through each of the scripts provided in the `IllustrativeAnalyses` folder.

8.1 Illustrative 0: Example data download

`illustrative_0_downloadExampleData.m`

DESCRIPTION

Due to their size, the EEG data files used in MatClassRSA illustrative analyses are not provided directly in the GitHub repository. Instead, they can be directly downloaded from a dedicated Stanford Digital Repository (SDR) dataset (B. C. Wang et al., 2025a).³² Thus, this first illustrative analysis script, `illustrative_0_downloadExampleData.m`, is not an analysis per se, but rather downloads essential large files used in the illustrative analyses. Specifically, the script does the following:

³²<https://purl.stanford.edu/kv831rr3606>

1. Locates the local `ExampleData` folder of the user's local instance of the MatClassRSA repository;
2. Determines whether each large data file in the SDR dataset is already in that folder; and
3. Downloads any files not found locally into the `ExampleData` folder and unzips if needed.

The script checks for the following files:

`S01.mat`, `S04.mat`, `S05.mat`, `S06.mat`, `S08.mat`. These files contain 124-channel EEG recorded, each recorded from a different adult participant. Each participant was shown 72 images drawn from 6 object categories in an ERP paradigm and viewed each image 72 times. For more information on the study design, see Kaneshiro et al. (2015b). In these data files, every observation represents a single trial. These files are directly the 'Data set 1' data analyzed in Kong et al. (2020), which themselves are re-cleaned versions of the data analyzed in Kaneshiro et al. (2015b). The original files that were cleaned for this version are from the OCED dataset (Kaneshiro et al., 2015a) and underwent a slightly different cleaning procedure than the Kaneshiro et al. (2015c) data analyzed in Kaneshiro et al. (2015b).

File information (per file):

- Data size: Approximately 200 MB
- Sampling rate: 62.5 Hz
- Epoch, relative to stimulus onset: -112 msec to 512 msec (40 time samples)
- Number of categories: 6 categories \times 12 exemplars per category = 72 exemplars
- Number of trials per exemplar: 72
- Number of trials per category: 864
- Total number of trials: 5184

Each file contains the following variables:

- `blCorrectIdx`: Vector of time indices to use for baseline-correcting the data (-112 to 64 msec).
- `fs`: Sampling rate of the data (62.5 Hz).
- `labels6`: Category-level labels of the trials. From 1 to 6, the ordered labels correspond to (Human Face-HF, Human Body-HB, Animal Face-AF, Animal Body-AB, Fruit/Vegetable-FV, Inanimate Object-IO) categories.
- `labels72`: Exemplar labels of the trials. The ordered labels correspond to the images in the *OCEDStimuli* folder.

- `subID`: Participant string identifier, e.g., 'S01' for participant 1.
- `t`: Vector of time indices, in msec, corresponding to time samples of data in `X`. Values range from -112 to 512 msec relative to stimulus onset.
- `X`: Matrix of cleaned data. Data dimensions are space-by-time-by-trial ($124 \times 40 \times 5184$).

OCEDStimuli.zip. This archive contains the 72 stimulus images corresponding to the above OCED .mat files. Numbering of the images corresponds to the elements of the `labels72` vector in the data files described above. Images are in .png format. If downloaded by this illustrative analysis script, the .zip archive will also be unzipped. Total file size is around 2 MB (zipped and unzipped). These images were subset from the database of 92 images that were published as part of Kriegeskorte et al. (2008b).

losorelli_100sweep_epoched.mat. This file, from the STAR-FFR-1 dataset (Losorelli et al., 2019), contains single-channel frequency-following response (FFR) EEG aggregated across 13 adult participants. Each participant was presented 6 short sounds representing speech syllables or synthesized musical notes; each stimulus was presented 2500 times to each participant. Due to constraints of the data-collection system, for this dataset each exported observation actually comprises 100 stimulus presentations ('sweeps') for a given participant and stimulus category. For more information on the study design, see Losorelli et al. (2020).

File information:

- Data size: 42 MB
- Sampling rate: 20 kHz
- Epoch, relative to stimulus onset: 5 msec to 145 msec (2801 time samples)
- Number of categories: 6
- Number of participants: 13
- Number of trials per category, across all participants: 325
- Total number of trials: 1950

The file contains the following variables:

- `P`: Vector of numeric per-trial participant identifiers, ranging from 1 to 13.
- `t`: Vector of time indices, in msec, corresponding to time samples of data in `X`. Values range from 5 to 145 msec relative to stimulus onset.
- `X`: Data matrix. Dimensions are trial-by-feature (1950×2801).
- `Y`: Trial labels. In order, stimulus labels represent spoken 'ba', 'da', and 'di' followed by synthesized piano, bassoon, and tuba instrument sounds.

Note: This file contains no variable denoting the sampling rate. As noted in Losorelli et al. (2020), and as can be calculated from the `t` vector, the data were sampled at 20 kHz.

MatClassRSA_v2_ExampleData_README.pdf. Informational document describing the dataset. By default, this file will be downloaded alongside the other data files into the user's local MatClassRSA directory.

Running the code

To run this script, the user should have their entire local MatClassRSA repository added to their MATLAB path. The function will then locate the `ExampleData` folder and designate that folder the destination for downloaded files. If more than one `MatClassRSA/ExampleData` directory is found in the user's MATLAB path, the script will print a warning and set the first indexed location as the download destination.

By default, the script will iterate through *all* files provided in the example data dataset (B. C. Wang et al., 2025a) and download any that are not already in the user's local directory. There is also an option for users to specify a subset of the available files to check for and download—for example, if interested in a specific illustrative analysis that does not use all of the data files. While not highlighted as a main option, users can also customize the script further to specify a different download destination.

8.2 Illustrative 1: Downstream impacts of preprocessing

`illustrative_1_impactOfPreprocessing.m`

DESCRIPTION

When working with multivariate M/EEG data, preprocessing decisions can substantially impact downstream outcomes such as classifier accuracies. This illustrative script explores how the `averageTrials()`, `noiseNormalization()`, and `shuffleData()` functions from the Preprocessing module can influence ERP waveforms, covariance matrices, and confusion matrices. First, we show how trial averaging improves EEG SNR, and that classifier accuracy can vary as a function of trial-averaging group size. Next, we show how noise normalization can change covariance matrix structure and perhaps negatively impact classifier performance for single-participant data, yet may improve generalizability of a classification when training and testing on data from different participants by reducing individual-specific noise structure—critical in cross-subject decoding contexts (Losorelli et al., 2020). Finally, we present a use case in which data shuffling can add value by distributing trials that are originally ordered by stimulus categories across the dataset prior to partitioning for cross validation.

Code and data

This analysis uses the following toolbox functions, in the following contexts:

- `Preprocessing.shuffleData()` — This function randomly shuffles class labels and/or participant labels, used to evaluate the effect of label structure on data shape and classification performance.
- `Preprocessing.noiseNormalization()` — Applies whitening to emphasize shared signal and reduce noise covariance structure, useful for within- and across-subject classification.
- `Preprocessing.averageTrials()` — Aggregates multiple trials into ‘pseudotrials’ to reduce noise and increase feature strength.
- `Classification.crossValidateMulti()` — Performs within-subject classification using cross-validation. Returns accuracy and confusion matrices.
- `Classification.trainMulti_opt()` — Trains an SVM classifier with optional hyperparameter tuning, by grid search.
- `Classification.predict()` — Applies a trained classifier to new data (either partitioned within-subject data or from a new subject) and returns predicted labels and confusion matrices.
- `Visualization.plotMatrix()` — Plots confusion matrices or representation dissimilarity matrices (RDMs) with optional group labeling and class-label color coding.
- `RDM_Computation.computeCMRDM()` — Computes representational dissimilarity matrices from confusion matrices.

In addition, this analysis uses the following data:

- `S01.mat`, `S04.mat`, `S05.mat`, `S06.mat`, `S08.mat` — EEG datasets from multiple participants (electrode \times time \times trial) as downloaded from the external dataset (B. C. Wang et al., 2025a) into the user’s local instance of MatClassRSA. Participant S01 is used for training, and the others for comparison and testing.
- `losorelli_100sweep_epoched.mat` — A two-dimensional EEG dataset (trial \times feature) used to demonstrate the effect of data and label shuffling on class- and participant-level averages. If not already on the user’s local machine, it can be downloaded by running the illustrative analysis `illustrative_0_downloadExampleData.m`.

For more information on the example data used in this illustrative example, see Chapter 8.1.

RESULTS AND INTERPRETATION

This illustrative analysis demonstrates how different preprocessing steps affect EEG decoding accuracy and data interpretability. In what follows, we present and interpret the outputs of trial averaging, noise normalization, and label shuffling procedures.

Trial averaging improves signal-to-noise ratio but reduces number of observations. We begin by visualizing how trial averaging sharpens ERP waveform structure across categories, by grouping 40 raw data trials into averaged ‘pseudotrials’.

Data Used

- Dataset: **S01.mat**
- Electrodes: **96**
- Time: **All**
- Trials: **All**
- Classes: **6-Class**

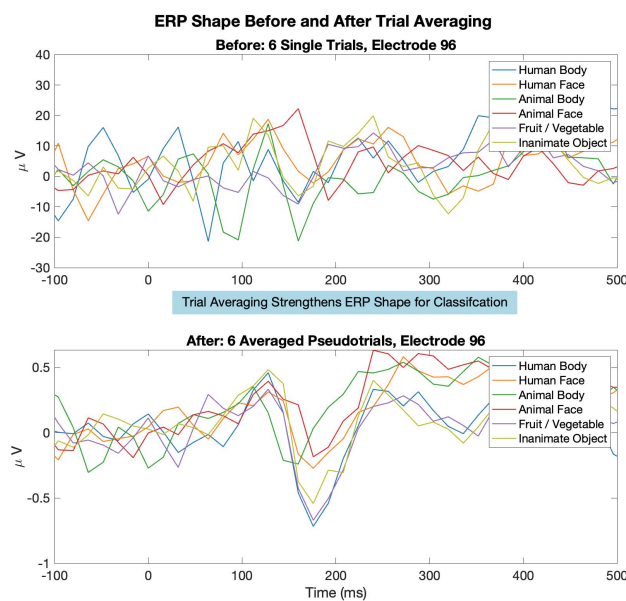


Figure 22: **ERP waveform before and after trial averaging.** Ten example single trials (top) and ten ‘pseudotrials’ formed via trial averaging of 40 trials (bottom) are shown for a representative electrode. Trial averaging reveals a clearer, more consistent ERP shape across categories.

As seen in Figure 22, trial averaging has a clear effect on the shape and stability of the ERP signal. In the top subplot, individual single-trial waveforms are noisy and variable across repetitions. However, in the bottom panel, averaging across trials results in clearer, more stereotyped ERP waveforms for each class, which also span a smaller range of voltages. This improves classification performance by enhancing signal-to-noise ratio. After first normalizing noise, and shuffling data, the trial-averaging step was implemented using the following function call:

```
[xAvg, yAvg] = Preprocessing.averageTrials(xNorm, yShuf, 40, ...  
'handleRemainder', 'append');
```

Classification accuracy varies by trial-averaging group size. Next, we explored how classification accuracy can vary according to the number of trials averaged together. Figure 23 shows the results of a sweep over different trial ‘group sizes’.

Data Used

- Dataset: **S01.mat**
- Electrodes: **96**
- Time: **144:304 msec**
- Trials: **All**
- Classes: **6-Class**

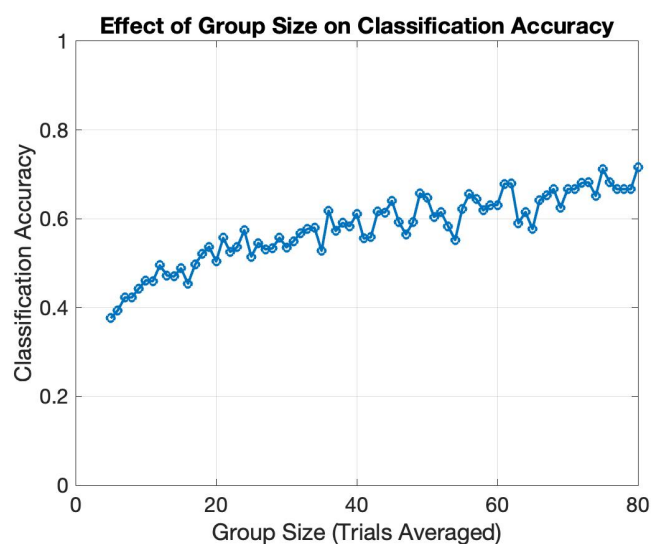


Figure 23: **Classification accuracy as a function of trial group size.** Accuracy improves as more trials are averaged together, stabilizing ERP representations and improving classification performance.

As the number of trials averaged into each ‘pseudotrial’ increases, classification accuracy steadily rises. This highlights a central trade-off in ERP decoding: While averaging more trials improves stability and decoding, it also reduces the number of available training samples. The curve suggests diminishing returns beyond a certain group size, emphasizing the need for dataset-specific tuning. The averaging and classification were performed using:

```
[xShuf, yShuf] = Preprocessing.shuffleData(X, labels6);  
xNorm = Preprocessing.noiseNormalization(xShuf, yShuf);  
[xAvg, yAvg] = Preprocessing.averageTrials(xNorm, yShuf, groupSize, [...]);  
M = Classification.crossValidateMulti(xAvg(e, t1:t2, :), yAvg);
```

where *e* is the electrode number being classified and *t1:t2* represent the range of time samples used.

Noise normalization impacts data covariance and classifier performance. We then examined the effects of noise normalization on electrode covariance structure and classification performance.

Data Used

- Dataset: **S01.mat**
- Electrodes: **All**
- Time: **All**
- Trials: **All**
- Classes: **6-Class**

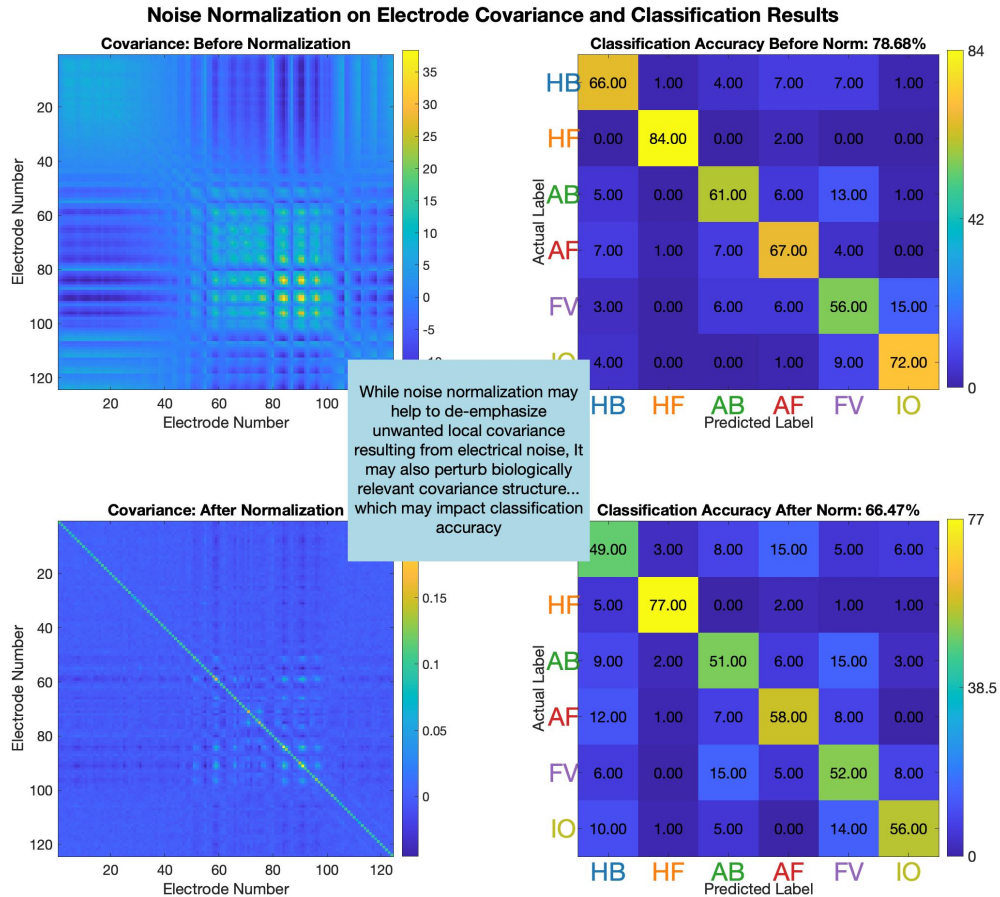


Figure 24: **Effect of noise normalization on covariance and classification accuracy.** Top: Covariance matrix and classification confusion matrix before normalization. Bottom: The same after noise normalization. Noise normalization alters inter-electrode covariance structure, which may influence classification differently depending on the nature of the data.

Figure 24 illustrates how noise normalization changes the covariance structure of the data and affects decoding. Before normalization (top row), inter-electrode covariance may reflect both neural signal and recording artifacts. After normalization (bottom row), the covariance matrix is more uniform, likely removing noise-related structure; however, potentially also removing biological relevant covariance structure. Despite that, classification results showed strong performance both with and without normalization. The normalization and classification steps were as follows:

```
[xShuf, yShuf] = Preprocessing.shuffleData(X, labels6, [...]);
```

```
[X_norm, sigma_inv] = Preprocessing.noiseNormalization(xShuf, yShuf);
```

```
[xAvgNorm, yAvgNorm] = Preprocessing.averageTrials(X_norm, yShuf, ...
```

```

10, 'handleRemainder', 'append', [...]);
[xAvg, yAvg] = Preprocessing.averageTrials(xShuf, yShuf, ...
10, 'handleRemainder', 'append', [...]);

MNorm = Classification.crossValidateMulti(xAvgNorm, yAvgNorm, ...
'classifier', 'SVM', 'PCA', 0.99, [...]);
M = Classification.crossValidateMulti(xAvg, yAvg, ...
'classifier', 'SVM', 'PCA', 0.99, [...]);

```

We then tested the effect of noise normalization on the generalizability of classification across participants.

Data Used

- Dataset: **S01.mat (training), S04.mat, S05.mat, S08.mat**
- Electrodes: **All**
- Time: **All**
- Trials: **All**
- Classes: **6-Class**

Figure 25 shows SVM confusion matrices for transfer learning—training on one participant and testing on others—before (top row) and after (bottom row) applying noise normalization. Without normalization, classification accuracy varies substantially across test participants, suggesting overfitting to subject-specific noise. After normalization, performance becomes more consistent and improves across the board. This indicates that noise normalization helps mitigate participant-specific idiosyncrasies. Training and testing involved:

```

[xShuf, yShuf] = Preprocessing.shuffleData(X, labels6, [...]);

[xAvg, yAvg] = Preprocessing.averageTrials(xShuf, yShuf, ...
30, 'handleRemainder', 'newGroup', [...]);

xNorm = Preprocessing.noiseNormalization(xShuf, yShuf);

M = Classification.trainMulti_opt(xAvg, yAvg, ...
'classifier', 'SVM', 'PCA', 0.99, [...]);
P = Classification.predict(M, xAvg, 'actualLabels', yAvg);

```

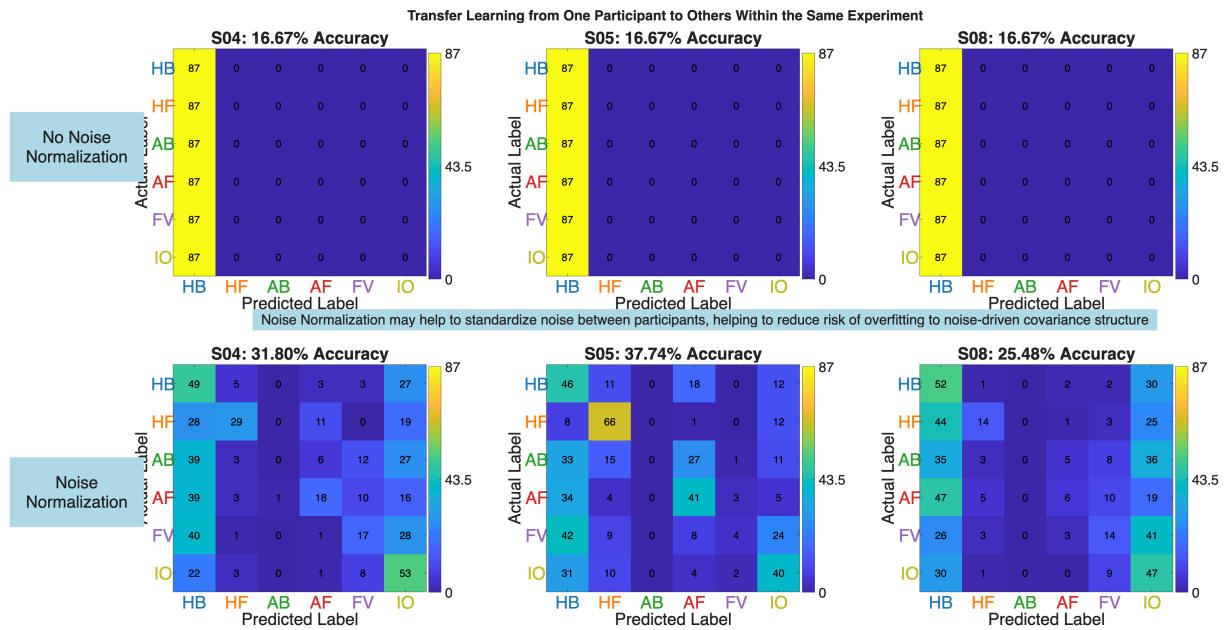


Figure 25: Effect of noise normalization on transfer learning across participants using SVM classifier. Each subplot shows the confusion matrix and classification accuracy when testing on a new participant, trained on a common source participant. Top row: Results without noise normalization. Bottom row: Results with normalization. Normalization appears to improve consistency and generalization across subjects.

To compare noise normalization effects across classifiers, we repeated the above analysis with the LDA classifier. As shown in Figure 26, the use of noise normalization again improves transfer learning. In this case, compared to SVM results (with default parameters; Figure 25), LDA performed slightly better both before and after the application of noise normalization.

Training and testing involved:

```
[xShuf, yShuf] = Preprocessing.shuffleData(X, labels6, [...]);

[xAvg, yAvg] = Preprocessing.averageTrials(xShuf, yShuf, ...
30, 'handleRemainder', 'newGroup', [...]);

xNorm = Preprocessing.noiseNormalization(xShuf, yShuf);

M = Classification.trainMulti_opt(xAvg, yAvg, ...
'classifier', 'LDA', 'PCA', 0.99, [...]);

P = Classification.predict(M, xAvg, 'actualLabels', yAvg);
```

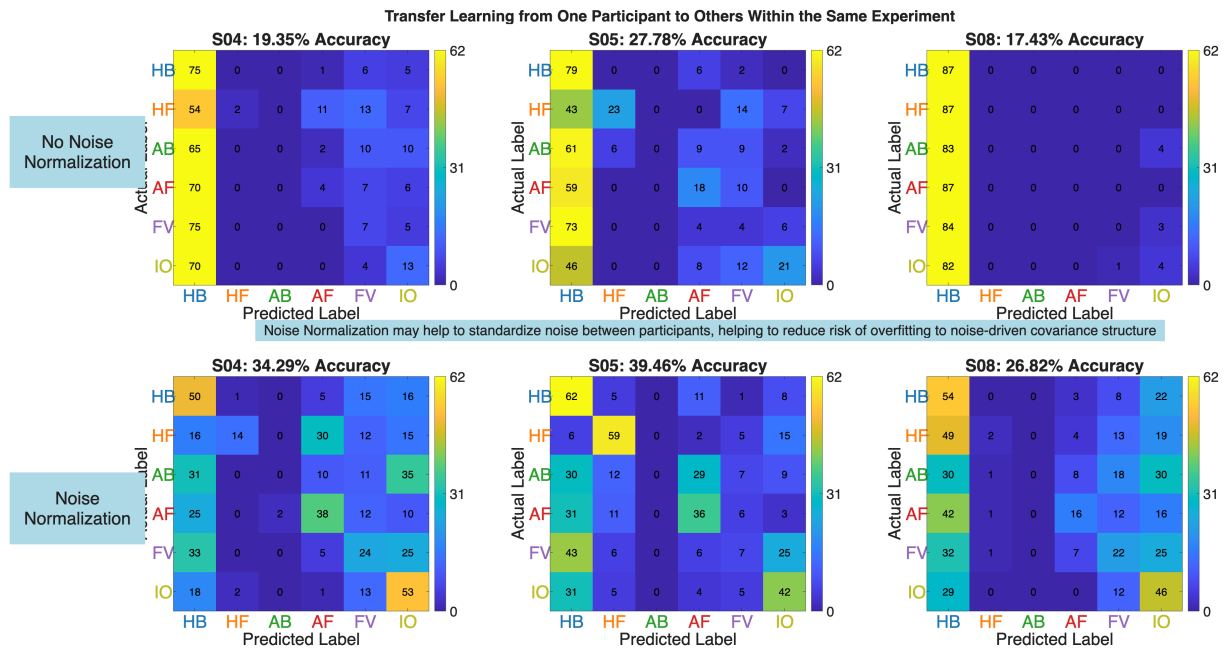


Figure 26: **Effect of noise normalization on transfer learning across participants using LDA classifier.** Each subplot shows the confusion matrix and classification accuracy when testing on a new participant, trained on a common source participant. Top row: Results without noise normalization. Bottom row: Results with normalization. Normalization appears to improve consistency and generalization across subjects.

Data shuffling distributes trials across train-test partitions while retaining trial and participant labels. In the next analysis, to ensure that models learn meaningful structure and not unintended regularities, we performed data shuffling controls.

Data Used

- Dataset: **losorelli_100sweep_epoched.mat**
- Time: **All**
- Trials: **All**
- Classes: **6-Class**

Figures 27 and 28 confirm that shuffling class labels, and respective data, randomizes class assignments without altering the underlying signal. Here, we expected the averaged data to be representative of the frequency-following response (FFR), encoding the stimulus fundamental frequency F0 of 100 Hz (Losorelli et al., 2020). These analyses are done as follows:

```
[X_shuf, Y_shuf, P_shuf, rndIdx] = Preprocessing.shuffleData(X, Y, P, [...]);
```

The grand-average FFR remains unchanged even after label shuffling, validating that classifiers rely on label-to-signal mappings, not trial order artifacts.

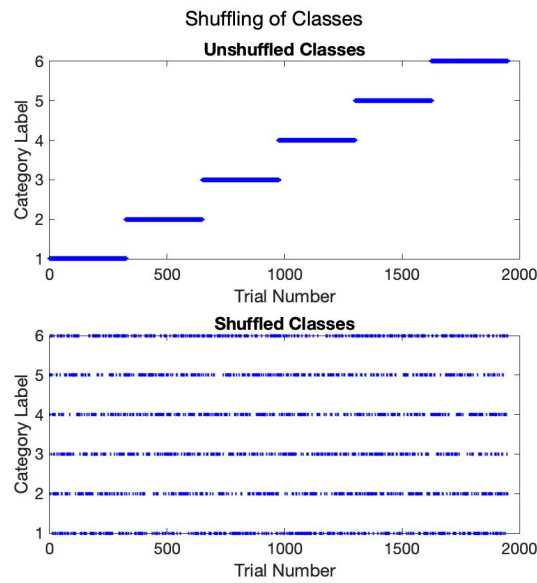


Figure 27: **Class labels before and after shuffling.** Shuffling class labels, and respective data, helps confirm that classification models are not exploiting unintended structure or trial-order artifacts.

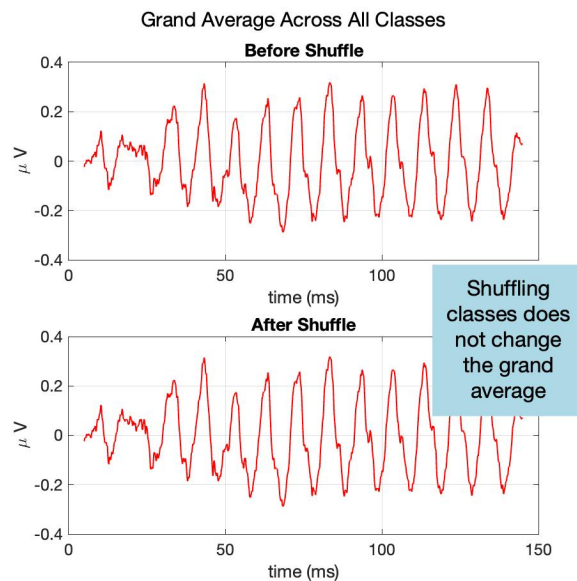


Figure 28: **Grand-average FFR before and after class label shuffling.** The average waveform for one condition remains unchanged by data shuffling, demonstrating that shuffling preserves FFR shape.

Lastly, we verified that participant identity did not bias classification by shuffling participant IDs. In Figures 29 and 30, participant IDs are randomized to prevent potential classification biases. This was implemented using:

```
[X_shuf, Y_shuf, P_shuf, rndIdx] = Preprocessing.shuffleData(X, Y, P, [...]);
```

Participant-specific FFR averages remain consistent post-shuffling, showing that while data are shuffled, the underlying signals remain intact.

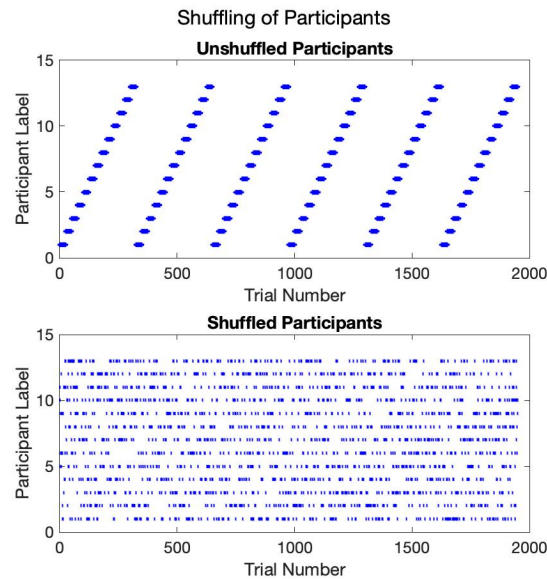


Figure 29: **Participant labels before and after shuffling.** Like class label shuffling, participant label shuffling removes unintended structure in the dataset that may bias classification results.

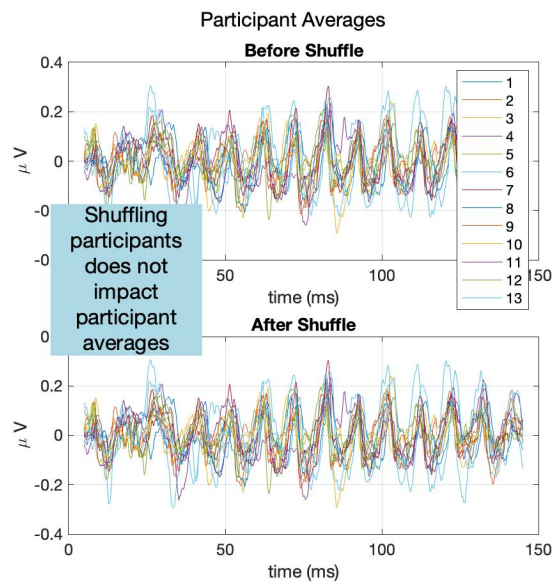


Figure 30: **Averaged FFR per participant before and after shuffling.** Participant-wise averages are preserved after shuffling participant indices and respective data, confirming that data structure remains intact.

DISCUSSION

In this example, we have explored the different Preprocessing module functions, highlighting cases in which they add value as well as cases where the user should examine impacts of their use in larger analysis pipelines. Trial averaging improves SNR, which can improve decoding but decreases the number of observations. Noise normalization may lead to mixed results but appears to be useful when training and testing a classifier across participants. Finally, shuffling trials can better distribute trials across cross-validation folds while maintaining linkages between trials, stimulus labels, and (if specified) participant identifiers.

8.3 Illustrative 2: Single-channel analyses

`illustrative_2_singleChannelAnalyses.m`

DESCRIPTION

Multichannel M/EEG datasets offer rich spatiotemporal information, but individual electrodes may vary in the reliability of the data they record, as well as their contributions to decoding performance. This illustrative analysis demonstrates how MatClassRSA functions can be used to assess reliability and classification outcomes at the single-sensor level. First, we consider single-channel reliability, plotting time-averaged reliabilities on a topomap and also plotting the time course of reliability at a single sensor. Similarly, we present a topomap of single-trial classifications. We also show how single-sensor classification outcomes may relate to a sensor's reliability metric. Finally, we assess relationships between single-sensor reliability and classification accuracy across a multi-channel dataset.

The analysis demonstrates how may relate to classification accuracy, highlighting the utility of reliability as a potential criterion for electrode selection. Such 'mass-univariate' approaches performed across a multi-sensor array can be viewed as complementary to spatial-filtering approaches and other strategies for dimensionality reduction while avoiding issues surrounding the direct visualization of spatial classifier weights (Haufe et al., 2014).

Code and data

This analysis uses the following toolbox functions, in the following contexts:

- `Reliability.computeSpaceTimeReliability()` — Computes split-half reliability of multichannel M/EEG data across electrodes and time. Used to identify sensors with stable, class-consistent signals across permutations.
- `Preprocessing.shuffleData()` — Randomly shuffles class labels. Used here prior to classification and averaging, to reduce temporal or order-based bias.

- `Preprocessing.noiseNormalization()` — Applies whitening to emphasize shared signal and reduce noise covariance structure, useful for within- and across-subject classification.
- `Preprocessing.averageTrials()` — Aggregates multiple trials into "pseudotrials" to reduce noise and increase feature strength.
- `Classification.crossValidateMulti()` — Performs cross-validated classification using a specified algorithm (e.g., LDA or SVM). Used for comparing decoding accuracy of individual electrodes.

In addition, this analysis uses the following data:

- `S01.mat` — A three-dimensional EEG dataset (electrode \times time \times trial) from participant 1. This dataset is included in the GitHub repository and is used here for within-subject classification and spatial reliability analysis. If not already on the user's local machine, it can be downloaded by running `illustrative_0_downloadExampleData.m`.
- `stimulus01.png`, ..., `stimulus62.png` — Visual stimulus exemplars associated with each class. These are used for visualization and are included in the GitHub repository.

For more information on the example data used in this illustrative example, see Chapter 8.1.

RESULTS AND INTERPRETATION

This illustrative analysis demonstrates how single-channel EEG classification and electrode reliability analysis can guide electrode selection for classification tasks. Through a series of figures, we show how spatial reliability varies, how it corresponds to decoding accuracy, and how classification performance varies across channels.

Reliable electrodes exhibit stable activity across trials. We began by computing reliability across electrodes and time points using split-half consistency.

Data Used

- Dataset: **S01.mat**
- Electrodes: **All**
- Time: **All**
- Trials: **All**
- Classes: **72-Class**

As seen in Figure 31, electrode reliability varies across the scalp. On the left, we visualize average reliability and standard deviation per electrode, and on the right, a topographic map shows spatial patterns in reliability. This was computed using:

```
reliability_time = Reliability.computeSpaceTimeReliability(...
X, labels72, [...]);
```

These reliability scores were averaged over time and across permutations.

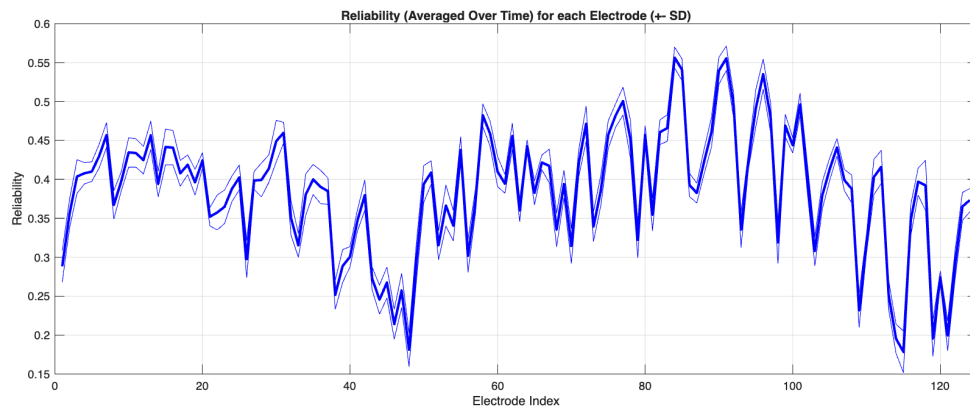


Figure 31: **Single-electrode reliabilities, averaged over all time points.** Mean and standard deviation of reliability for each electrode across permutations.

Next, we plotted the time-resolved reliability of a highly reliable (across all time) electrode. Figure 32 shows the mean reliability over time for electrode 96, the most reliable channel identified in the previous analysis. The stable reliability beginning around 80 msec after stimulus onset suggests that this sensor may be useful in classification. The code used was:

```
reliability_96 = squeeze(reliability_time(96, :, :));
```

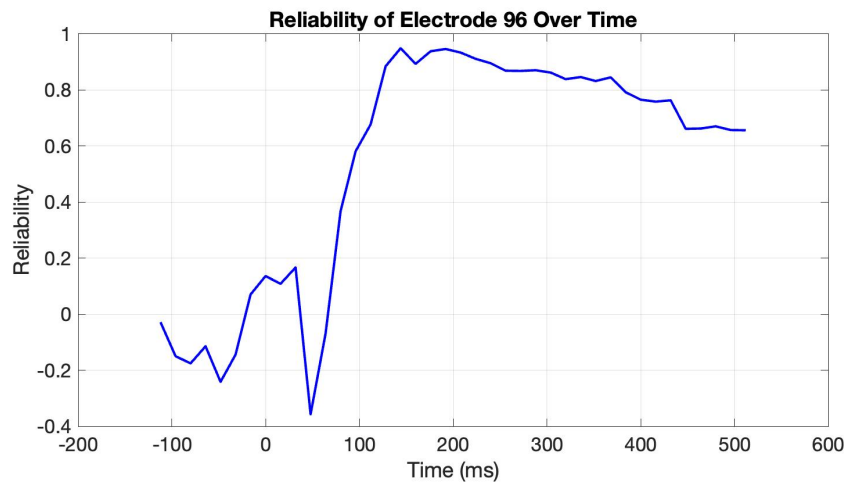


Figure 32: **Reliability of electrode 96 over time.** Electrode 96 shows stable high reliability across latencies associated with visual processing (from around 80 msec post-stimulus onset), supporting its potential value for classification.

Single-sensor reliability may predict classification performance. We then evaluated the effect of using a reliable vs. unreliable electrode for classification, using LDA and confusion matrices. For this analyses we used a subset of time points, focusing on the latencies near the N170 component:

Data Used

- Dataset: **S01.mat**
- Electrodes: **96, 48**
- Time: **128:224 msec**
- Trials: **All**
- Classes: **6-Class**

Figure 33 demonstrates that classification accuracy is noticeably higher for a reliable electrode (left) compared to an unreliable one (right). Noise normalization and trial averaging were applied before classification:

```
xNorm = Preprocessing.noiseNormalization(xShuf,yShuf);
[xAvg, yAvg] = Preprocessing.averageTrials(xNorm(elec, t1:t2, :), ...
yShuf, 15);
M = Classification.crossValidateMulti(xAvg, yAvg);
```

where `elec` is the electrode number being classified and `t1:t2` represent the range of time samples used. This shows the direct benefit of data reliability in model performance.

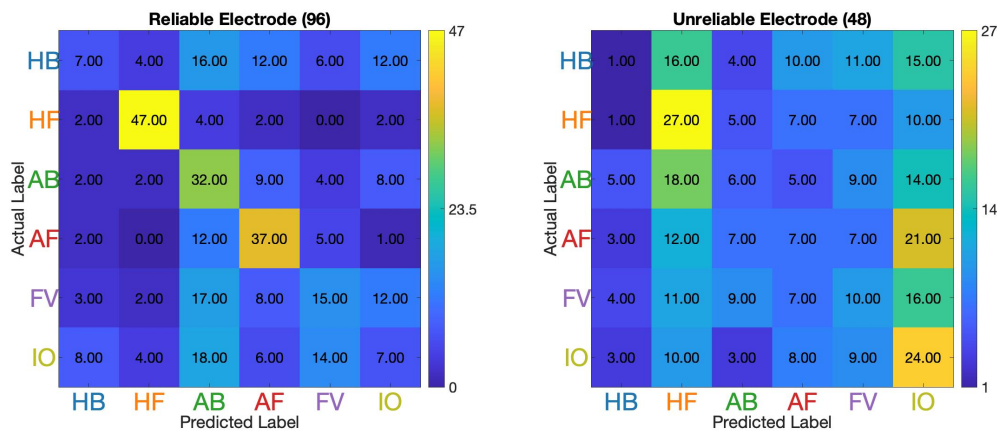


Figure 33: **Confusion matrices for reliable versus unreliable electrodes.** Left: Electrode 96. Right: Electrode 48. Classification was performed using data points 128–224 msec post-stimulus onset. The reliable channel produces higher classification accuracy.

We extended this investigation to all electrodes and all time points to evaluate the relationship between classification accuracy and reliability. For these examples we use a general-purpose topoplot function, `topoplotStandalone()`, which is located in the `+Utils` folder. This function was developed by the Parra Lab³³ and is publicly available on GitHub as part of the codebase for Dmochowski et al. (2018).³⁴ It has the same functionalities as the EEGLAB topoplot function³⁵ but is self-contained in the sense that it has no EEGLAB dependencies. For the present examples we have created a `locsEGI124.mat` file (also in `+Utils`) to store the 124-channel sensor locations as a variable. However, in general, users can render their own topoplots by inputting the filename of their preferred locations file to the `topoplotStandalone()` function.

Data Used

- Dataset: **S01.mat**
- Electrodes: **Individual**
- Time: **All**
- Trials: **All**
- Classes: **6-Class**

As shown in Figure 34 and Figure 35, there is a broad correspondence between an electrode's average reliability and its classification accuracy. The best performing classifier (electrode

³³<https://parralab.org/>

³⁴https://github.com/dmochow/SRC/blob/master/topoplot_new.m

³⁵<https://scn.ucsd.edu/~arno/eeglab/auto/topoplot.html>

96) is among the highest in reliability. While the relationship is not perfect, it suggests that reliability estimates can provide a useful prior when selecting sensors for analysis. This relationship was visualized using:

```
Xsingle = X(elec, :, :);
xNorm = Preprocessing.noiseNormalization(Xsingle, labels6);

M = Classification.crossValidateMulti(xNorm, labels6);

avg_space_reliability_space = squeeze(mean(reliability_time, 2));

montage = load('+Utils/locsEGI124.mat');

Utils.topoplotStandalone([mean(avg_space_reliability_space, 2); nan(4,1)], ...
    montage.locs, 'electrodes', 'labels', [...]);

Utils.topoplotStandalone(accuracies, ...
    montage.locs, 'electrodes', 'labels', [...]);
```

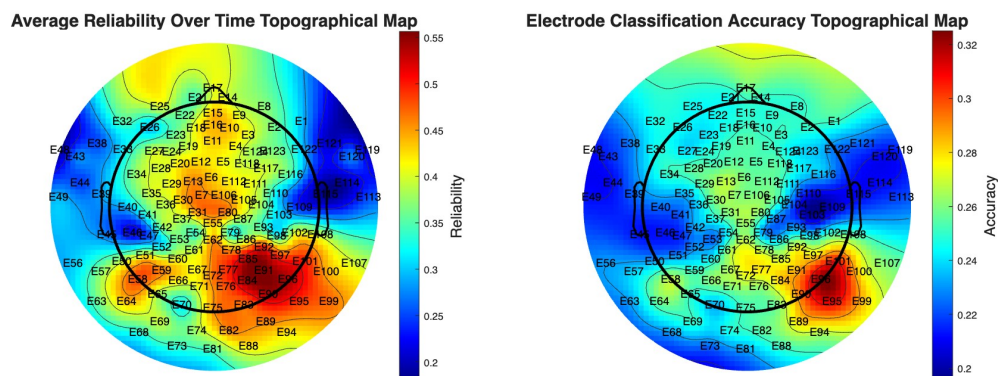


Figure 34: **Topographical Map: Single-electrode classification accuracy and reliability.** The single-electrode classification accuracy and reliability (using all available time points), overlaid on electrode montage, shows markedly similar topographies.

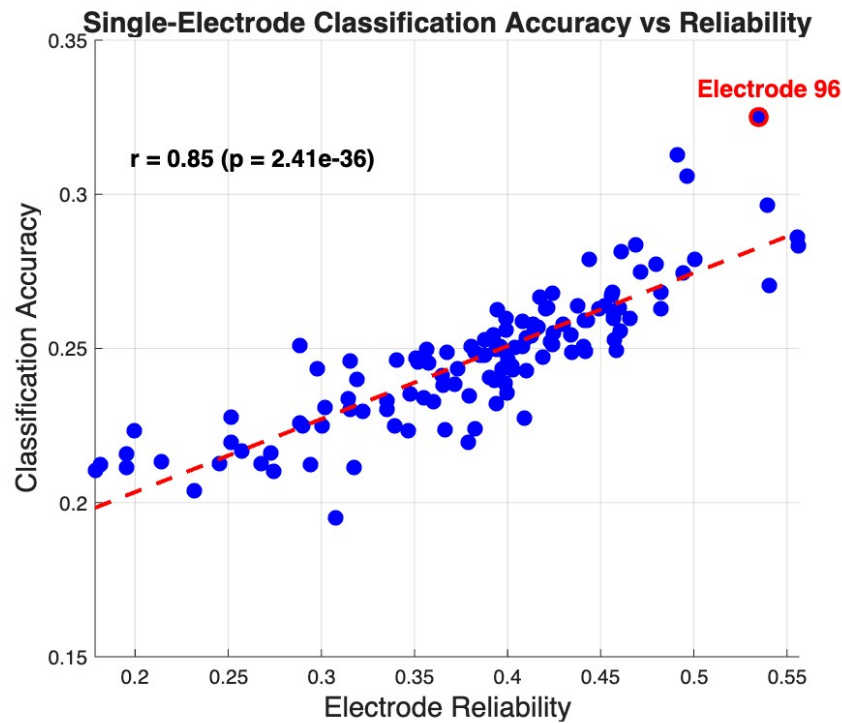


Figure 35: **Single-electrode classification accuracy versus reliability.** There is a clear positive correlation between single-electrode classification accuracy and electrode reliability ($r=0.85$, $p=2.41e-36$), suggesting electrode reliability explains ~72% of the variance. Taken together, reliability seems to be a strong predictor of classification accuracy, where more reliable electrodes yield higher accuracies.

DISCUSSION

This example has illustrated how single-channel reliability can vary across the electrode montage, and that the reliability of data recorded by a given electrode may predict its decoding performance to some extent:

- Electrode 96 was identified as a highly reliable sensor through split-half reliability computations.
- Reliability of electrode 96 was consistently high starting around 80 msec after stimulus onset.
- Classification using this electrode outperformed classification using a low-reliability electrode.
- Across all electrodes, reliability was correlated with decoding accuracy; hence, reliability could be a useful metric if needing to select a subset of electrodes for classification.

In all, while classification on its own will derive optimal weightings of multi-channel data, single-channel analyses may also be a useful way to identify relevant spatial components of the data. Split-half reliability may be suggestive of single-electrode classifier performance.

8.4 Illustrative 3: Time-resolved analyses

`illustrative_3_timeResolvedAnalyses.m`

DESCRIPTION

The high temporal resolution of M/EEG data lends itself to time-resolved analyses. This illustrative script demonstrates how users can use functions from the Reliability and Classification modules to identify time windows within a response epoch that are the most e.g., reliable or informative for decoding. The analyses in this section include calculation of reliability over time, classification accuracy as a function of time window size, and relationships between time-resolved classifier accuracy and ERP waveforms. We conclude by demonstrating that classifying optimal time windows of the response can improve classifier accuracy as well as category separability in downstream visualizations.

Code and Data

This analysis uses the following toolbox functions, in the following contexts:

- `Preprocessing.shuffleData()` — Randomly shuffles class labels. Used here prior to classification and averaging, to reduce temporal or order-based bias.
- `Preprocessing.noiseNormalization()` — Applies whitening to emphasize shared signal and reduce noise covariance structure, useful for within- and across-subject classification.
- `Preprocessing.averageTrials()` — Aggregates multiple trials into "pseudotrials" to reduce noise and increase feature strength.
- `Reliability.computeSpaceTimeReliability()` — Estimates split-half reliability of ERP data across electrodes and time.
- `Classification.crossValidateMulti()` — Performs multiclass classification using cross-validation. Returns accuracy and confusion matrices.
- `RDM_Computation.computeCMRDM()` — Computes representational dissimilarity matrices (RDMs) from confusion matrices.
- `Visualization.plotMatrix()`, `plotDendrogram()`, and `plotMDS()` — Visualize confusion matrices and derived representational geometry.

This analysis uses the following dataset:

- **S01.mat** — A three-dimensional EEG dataset (electrode \times time \times trial) from one participant. Included in the GitHub repository. Contains responses to 72 stimuli grouped into 6 classes.

For more information on the example data used in this illustrative example, see Chapter 8.1.

RESULTS AND INTERPRETATION

Time-resolved reliability peaks in early ERP window. We first examined time-resolved reliability to identify windows of stable signal.

Data Used

- Dataset: **S01.mat**
- Electrodes: **All**
- Time: **All**
- Trials: **All**
- Classes: **6-Class and 72-Class**

As seen in Figure 36, reliability of the evoked response is highest between 130 and 250 msec. Moreover, time-resolved reliability is more stable when considering 72-class (exemplar) rather than 6-class (category) trial labels; this is likely because the reliability calculation involves correlations of the data across all of the available labels. In all, measures of reliability over time likely reflect response latencies relevant for distinguishing stimulus categories. This was visualized using:

```
reliability_time_6 = Reliability.computeSpaceTimeReliability(...  
X, labels6, 'numPermutations', n_perm, [...]);  
  
reliability_time_72 = Reliability.computeSpaceTimeReliability(...  
X, labels72, 'numPermutations', n_perm, [...]);
```

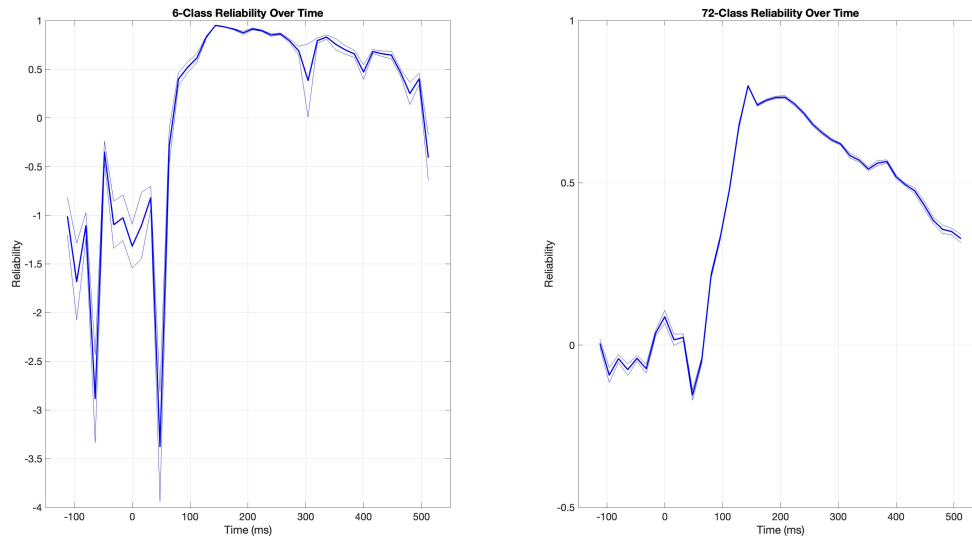


Figure 36: **Time-resolved reliability (6-class and 72-class).** Reliability over time averaged across electrodes, for 72-class (left) and 6-class (right).

ERP morphology differs between classes in the reliable window. We next visualized ERPs for each class to assess differences that might support classification.

Data Used

- Dataset: **S01.mat**
- Electrodes: **96**
- Time: **All**
- Trials: **All**
- Classes: **6-Class**

In Figure 37, we see that class-mean ERPs begin to diverge within the 100–300 msec window, consistent with the reliability peak. This supports using this window for time-resolved classification.

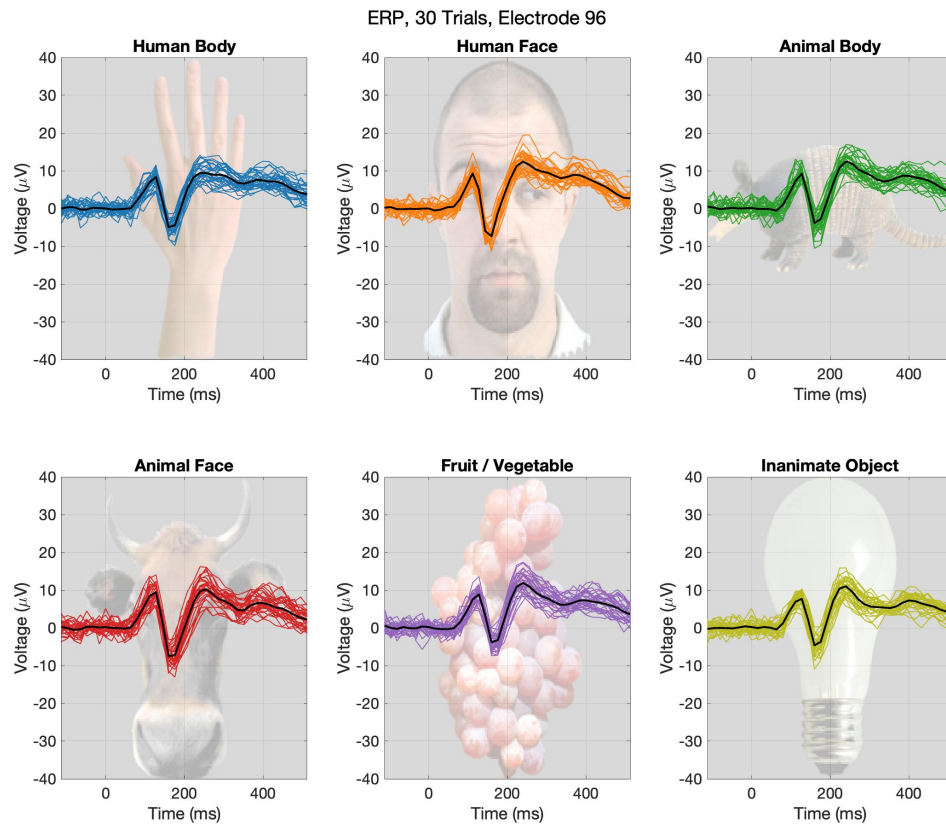


Figure 37: **ERPs for each stimulus class at well-classifying electrode.** Each subplot shows 30 pseudotrials (faint) and the average ERP over all pseudotrials (bold) for channel 96, overlaid on the class stimulus image. Categories diverge in the same time window observed in the reliability analysis.

Decoding varies by length of time window. In ERP analysis, temporal precision is critical because different experimental conditions often evoke subtle but time-locked differences in the evoked response. We conducted sliding-window classification with variable time-window sizes to identify the optimal temporal resolution for decoding.

Data Used

- Dataset: **S01.mat**
- Electrodes: **All**
- Time: **Selected Bin**
- Trials: **All**
- Classes: **6-Class**

Figure 38 shows that the maximum decoding accuracy increases with the length of the time bin, up to a peak at around 6 samples. This may reflect better integration of temporally distributed ERP features. As Figure 38 suggests, smoothing across a time bin too large could obscure these features. These classifications were computed by an LDA classifier, in the following way:

```
[xShuf, yShuf] = Preprocessing.shuffleData(X, labels6, [...]);
xNorm = Preprocessing.noiseNormalization(xShuf, yShuf);
[xAvg, yAvg] = Preprocessing.averageTrials(xNorm, yShuf, 15);
M = Classification.crossValidateMulti(Xbin, yAvg, [...]);
```

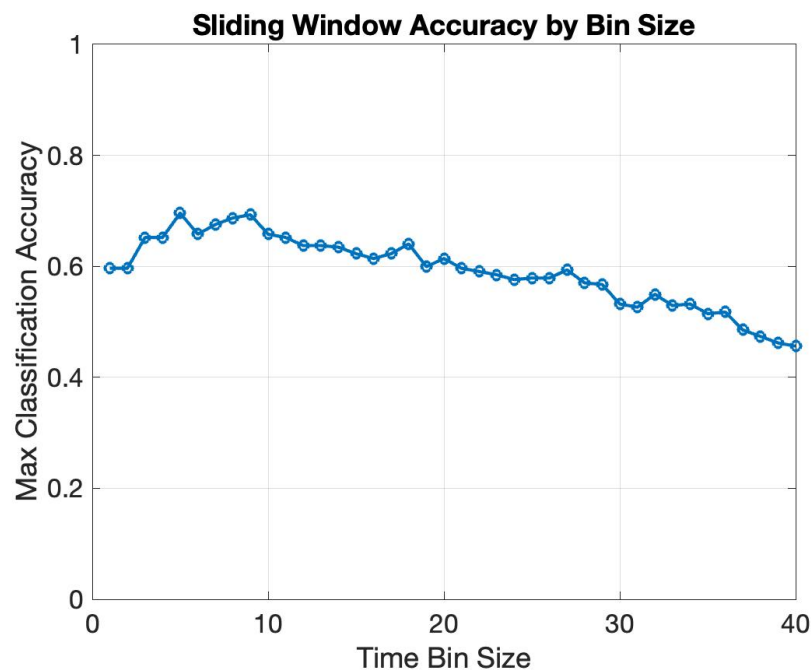


Figure 38: **Maximum classification accuracy by varying time bin size.** Sliding-window 6-class classifications (chance-level accuracy: 16.67%) were run across the entire epoch at bin sizes ranging from 1 to 40 samples, where each sample represents 16 msec. The maximal classification accuracy for each bin size is reported in the figure. All bin sizes produce maximal accuracy well above chance level. The accuracy plateau between 6–15 samples suggests this range may preserve key ERP waveform features. Larger bins correspond to slightly lower classification accuracy, perhaps due to inclusion of data from time intervals during which stimulus categories are less separable.

P1–N1 ERP peaks support classification. Next, we selected a bin size of 6 and visualized accuracy over time. Figure 39 shows that peak classification performance corresponds to the P1 and N1 components (~128–208 msec). This confirms the value of using both reliability

and ERP shape to guide the selection of temporal decoding windows. The same classification results from the previous figure were used to generate this figure.

These comparisons were created using the following code:

```
[xShuf, yShuf] = Preprocessing.shuffleData(X, labels6, [...]);
xNorm = Preprocessing.noiseNormalization(xShuf, yShuf);
[xAvg, yAvg] = Preprocessing.averageTrials(xNorm, yShuf, 15);

M = Classification.crossValidateMulti(Xbin, yAvg, [...]);
accs(bin) = M.accuracy;
```

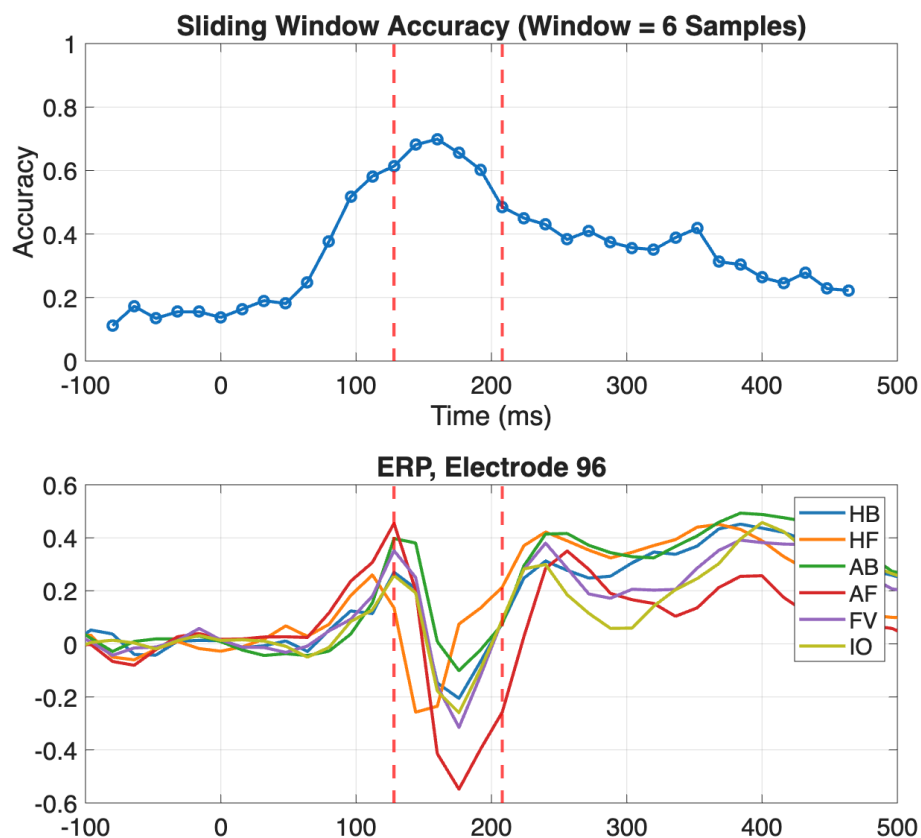


Figure 39: **Sliding window classification accuracy and ERP.** Top: Decoding accuracy over time using 6-sample bins. Bottom: ERP per class. The peak decoding aligns with the ERP P1 and N1 components (highlighted by vertical red lines).

Focusing on reliable timepoints improves accuracy. We then compared confusion matrices computed using all timepoints vs. just the high-performing time window. In Figure 40, classification accuracy (chance level 16.67%) improves from 42.69% when the model is trained on all time points to 69.88% when the model is trained only on the time window identified by the reliability and ERP analysis. This suggests that less stimulus-driven relevant timepoints may add noise. The following code was used for this analysis:

```
[xShuf, yShuf] = Preprocessing.shuffleData(X, labels6, [...]);
xNorm = Preprocessing.noiseNormalization(xShuf, yShuf);
[xAvg, yAvg] = Preprocessing.averageTrials(xNorm, yShuf, 15);

MAll = Classification.crossValidateMulti(xAvg, yAvg, [...]);
MBest = Classification.crossValidateMulti(xAvg(:, startT:endT, :), yAvg, [...]);
```

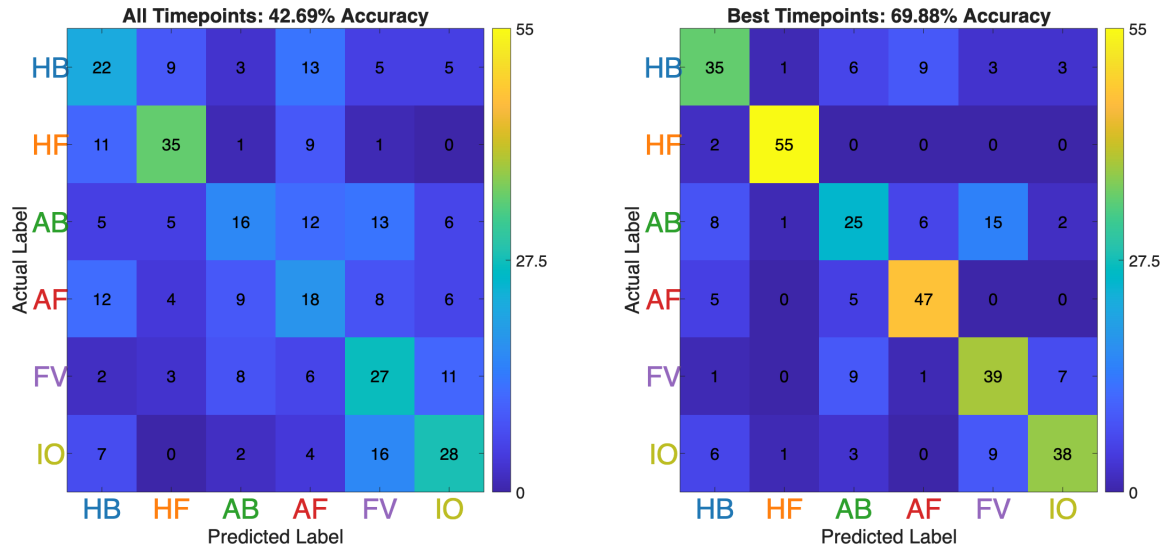


Figure 40: **Confusion matrices: All timepoints versus well-performing temporal subset.** Left: Classifier trained on all timepoints. Right: Classifier trained on 128–208 msec. Performance improves when restricting to informative timepoints.

Time selection clarifies category relationships. We then computed representational dissimilarity matrices (RDMs) and plotted dendrograms and MDS. As seen in Figure 41, categories separate more cleanly in the dendrogram derived from the selected time window. This reflects greater internal consistency and separability. This dendrogram visualization was done in the following way:

```
Visualization.plotDendrogram(RDMA11, ...
'nodeLabels', catLabels, 'nodeColors', rgb6);

Visualization.plotDendrogram(RDMBest, ...
'nodeLabels', catLabels, 'nodeColors', rgb6);
```

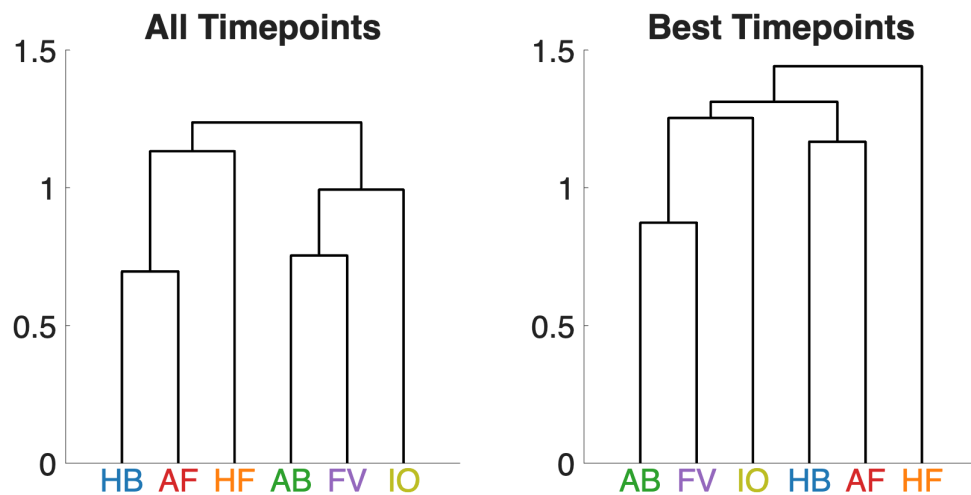


Figure 41: **Dendrograms: All timepoints versus well-performing temporal subset.** Left: Dendrogram computed from full-time classifier. Right: Dendrogram from the restricted time window (128–208 msec). Hierarchical relationships more closely reflect the a priori category structure in the more informative window.

We also visualized class similarity using MDS plots. As in Figure 42, MDS coordinates of inter-class distances change when temporally focused classification is performed. The MDS plots were generated in the following way:

```
Visualization.plotMDS(RDMA11, ...
'nodeLabels', catLabels, 'nodeColors', rgb6);
```

```
Visualization.plotMDS(RDMBest, ...
'nodeLabels', catLabels, 'nodeColors', rgb6);
```

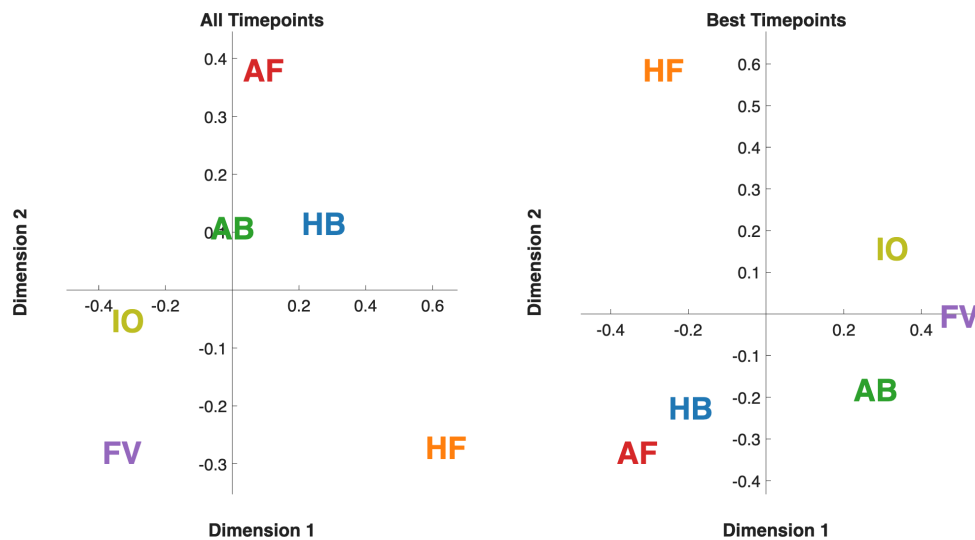


Figure 42: **MDS plots: All timepoints versus well-performing temporal subset.** Left: All timepoints. Right: Best timepoints (128–208 msec). When all timepoints are used, MDS Dimension 1 broadly separates the categories into FV/IO, AB/AF, HB, and HF groups while Dimension 2 further separates IO from FV and AF from AB. When best timepoints are used, the categories separate differently along these two dimensions (e.g., HF is now separated along Dimension 2).

DISCUSSION

Complementing the single-channel analyses, this example demonstrates how reliability and decodability of the data can vary over time, and how to apply these approaches to subsets of the response epoch to identify informative temporal windows.

- Time-resolved reliability clarifies what portions of the response epoch are stable and consistent across trials.
- ERP visualizations further highlight portions of the response epoch with meaningful signal (128–208 msec).
- Sliding-window classification shows that decoding also varies temporally over the response epoch, and may also be impacted by the length of the window used.
- Restricting classification to the optimal time window improves confusion matrix separability, as well as RDM, dendrogram, and MDS clarity.

Together, these results support the use of time-resolved reliability and classification as data-driven strategies for identifying meaningful epochs in the data, as well as to connect classification results to e.g., established ERP components.

8.5 Illustrative 4: Train-test SVM optimization

`illustrative_4_trainTestOptimizeSVM.m`

DESCRIPTION

This illustrative script demonstrates how iterative hyperparameter optimization of MatClassRSA's SVM classification functions can further improve classifier performance. We refine the SVM model by narrowing the search space for the 'C' and 'gamma' parameters over iterative optimizations.

Code and data

This analysis uses the following toolbox functions in the following contexts:

- `Preprocessing.shuffleData()` — Randomly shuffles class labels. Used here prior to classification and averaging, to reduce temporal or order-based bias.
- `Preprocessing.noiseNormalization()` — Applies whitening to emphasize shared signal and reduce noise covariance structure, useful for within- and across-subject classification.
- `Preprocessing.averageTrials()` — Aggregates multiple trials into "pseudotrials" to reduce noise and increase feature strength.
- `Classification.crossValidateMulti_opt()` — Performs grid search-based SVM classification.
- `Visualization.plotMatrix()` — Plots confusion matrices with color and label metadata.

In addition, this analysis uses the following data:

- `S01.mat` — A three-dimensional EEG dataset (electrode \times time \times trial) from one participant. Included in the GitHub repository. Contains responses to 72 stimuli grouped into 6 classes.

For more information on the example data used in this illustrative example, see Chapter 8.1.

RESULTS AND INTERPRETATION

Initial classification: Default grid search of hyperparameters gamma and C. We begin by performing default grid search-based SVM classification using `Classification.crossValidateMulti_opt` on subject S01.

Data Used

- Dataset: **S01.mat**
- Electrodes: **All**
- Time: **All**
- Trials: **All**
- Classes: **6-Class**

The SVM model is built using default parameters of an RBF-kernel SVM trained using default grid search (5 logarithmically spaced points between 10^{-5} and 10^5) and 10-fold cross validation, as follows:

```
[xShuf, yShuf] = Preprocessing.shuffleData(X, labels6, [...])
xNorm = Preprocessing.noiseNormalization(xShuf, yShuf);
[xAvg, yAvg] = Preprocessing.averageTrials(xNorm, yShuf, ...
10, 'handleRemainder', 'newGroup', [...]);
```

```
MSVM = Classification.crossValidateMulti_opt(xAvg, yAvg, ...
'classifier', 'SVM', 'kernel', 'rbf', 'PCA', 0.99);
```

The resulting confusion matrix is shown in Figure 43; here the SVM model achieves a modest accuracy of 66.48% for optimal gamma of 0.0032 and optimal C of 316.

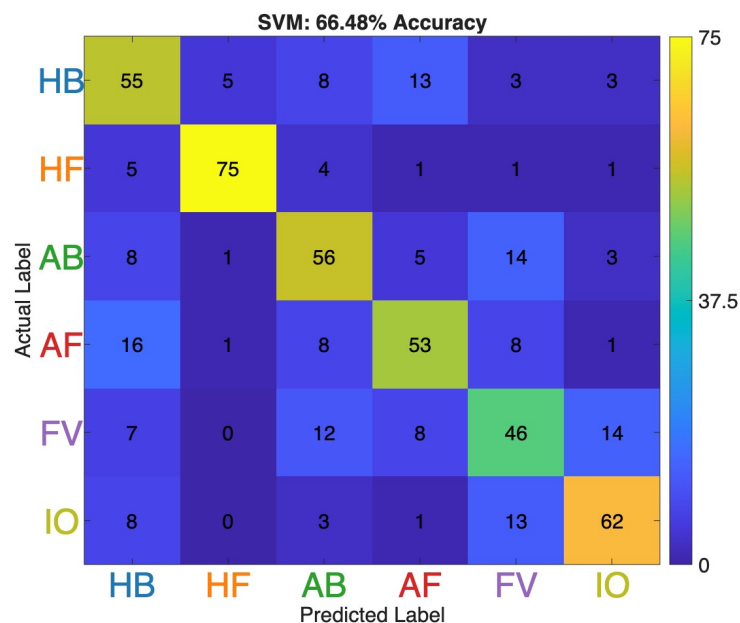


Figure 43: **SVM with default grid search hyperparameters.** Default hyperparameters (logspaced) and 10-fold cross validation yield a 6-class classification accuracy of 66.48%. Grid search identified optimal gamma of 0.0032 and optimal C of 316. (There is a known issue

Refining the search grid improves SVM performance. To determine whether further hyperparameter tuning could produce better classifier performance, we then refined the SVM grid search to optimize the gamma and C hyperparameters over smaller ranges of values that are closer to the previously identified optima. This search, again with 10-fold cross validation, is implemented as follows:

```
'gammaSpace', logspace(-1.8, -3, 20)
'CSpace', logspace(3, 5, 20)

MSVM = Classification.crossValidateMulti_opt(xAvg, yAvg, ...
      'classifier', 'SVM', 'kernel', 'rbf', 'PCA', 0.99, 'gammaSpace', ...
      gamma_fine, 'CSpace', C_fine);
```

Figure 44 shows the confusion matrix after restricting the SVM grid search to a smaller range around the previously identified gamma = 0.0032 and C = 316. This follow-up attempt identifies optimal gamma = 0.0146 and C = 1000, with a corresponding accuracy of 81.23%.

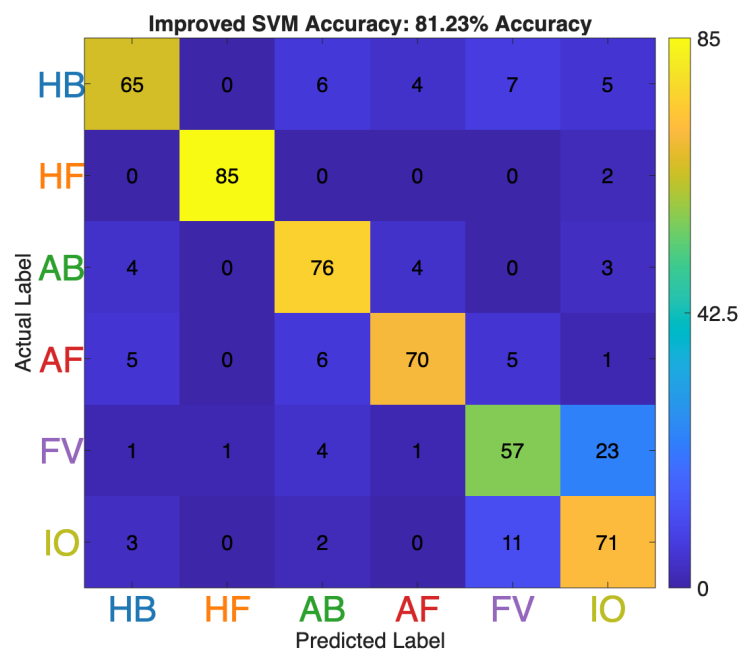


Figure 44: **Refined SVM grid search.** Focusing on a narrower range near the previously identified best values yielded improved accuracy. This refined grid search identified optimal gamma of 0.0146 and optimal C of 1000, and 10-fold cross-validated accuracy of 81.23%.

DISCUSSION

This analysis highlights the value of hyperparameter optimization when using SVM for M/EEG classification. As the selection of the 'C' and 'gamma' hyperparameter values can

impact classification performance, iterative optimization can further improve performance over MatClassRSA's default grid-search values.

- Using default grid search, SVM classification yields moderate accuracy.
- By narrowing the search space based on prior results, we improved classification accuracy substantially.

Taken together, these results emphasize the importance of hyperparameter tuning when performing SVM classifications.

8.6 Illustrative 5: Compare different RDM constructions

`illustrative_5_compareRDMs.m`

DESCRIPTION

In this analysis, we compare representational dissimilarity matrices (RDMs) computed from different classifier outputs as well as directly from the original data. The script is structured into three main sections: (1) Comparing RDMs derived from multi-class confusion matrices versus pairwise accuracy matrices; (2) comparing Pearson versus Euclidean RDMs at a specific electrode, and (3) comparing those same distance metrics at a specific time point.

Code and Data

- `Preprocessing.shuffleData()` — Shuffles labels to mitigate ordering bias.
- `Preprocessing.noiseNormalization()` — Applies whitening to reduce noise-based covariance.
- `Preprocessing.averageTrials()` — Creates pseudotrials by averaging to boost signal.
- `Classification.crossValidateMulti()` — Performs multi-class classification with LDA.
- `Classification.crossValidatePairs()` — Computes pairwise LDA classification between all class pairs.
- `RDM_Computation.computeCMRDM()` — Computes RDM from confusion matrix.
- `RDM_Computation.shiftPairwiseAccuracyRDM()` — Transforms pairwise accuracy into RDM format.
- `RDM_Computation.computePearsonRDM()`, `computeEuclideanRDM()` — Compute dissimilarity matrices from raw EEG based on correlation or Euclidean distance.

- `Visualization.plotMatrix()`, `plotMDS()`, `plotDendrogram()`, `plotMST()` — RDM visualization tools.

This analysis uses the following dataset:

- **S06.mat** — A three-dimensional EEG dataset (electrode \times time \times trial) from one participant. Included in the GitHub repository. Contains responses to 72 stimuli grouped into 6 classes.

For more information on the example data used in this illustrative example, see Chapter 8.1.

RESULTS AND INTERPRETATION

In this illustrative analysis, we explored multiple forms of representational dissimilarity matrices (RDMs) computed from EEG data. We visualized how different classification and RDM construction approaches reveal distinct structures in the data.

Multi-class and Pairwise Classification. We first compared RDMs derived from multi-class classification and pairwise classification using the same dataset (subject S06, all electrodes and timepoints). This analysis used the following code:

```
----- Preprocessing-----
[xShuf, yShuf] = Preprocessing.shuffleData(X, labels6, [...])
xNorm = Preprocessing.noiseNormalization(xShuf, yShuf);
[xAvg, yAvg] = Preprocessing.averageTrials(xNorm, yShuf, 20, ...
    'handleRemainder', 'newGroup' [...]);

----- Classification-----
MMC = Classification.crossValidateMulti(xAvg, yAvg, ...
    'classifier', 'LDA', 'PCA', 0.99);

MPW = Classification.crossValidatePairs(xAvg, yAvg, ...
    'classifier', 'LDA', 'PCA', 0.99);

----- RDM Computation-----
[RDMmc, params] = RDM_Computation.computeCMRDM(MMC.CM);

RDMpm = RDM_Computation.shiftPairwiseAccuracyRDM(MPW.AM);

----- Visualization-----
Visualization.plotMatrix(RDMmc, 'colorbar', 1, 'matrixLabels', 1, ...
    'rankType', 'p', [...]);
```

```
Visualization.plotMatrix(RDMpm, 'colorbar', 1, 'matrixLabels', 1, ...  
    'rankType', 'p', [...]);
```

As seen in Figure 45, the RDM, MDS, and dendrogram visualizations look broadly similar across the multiclass and pairwise classification approaches. The main difference is that the MDS and dendrogram plots, which are constructed based on the actual distances and not the percentile ranks shown in the RDM plots, display larger distances overall for the multiclass results compared to the pairwise accuracies.

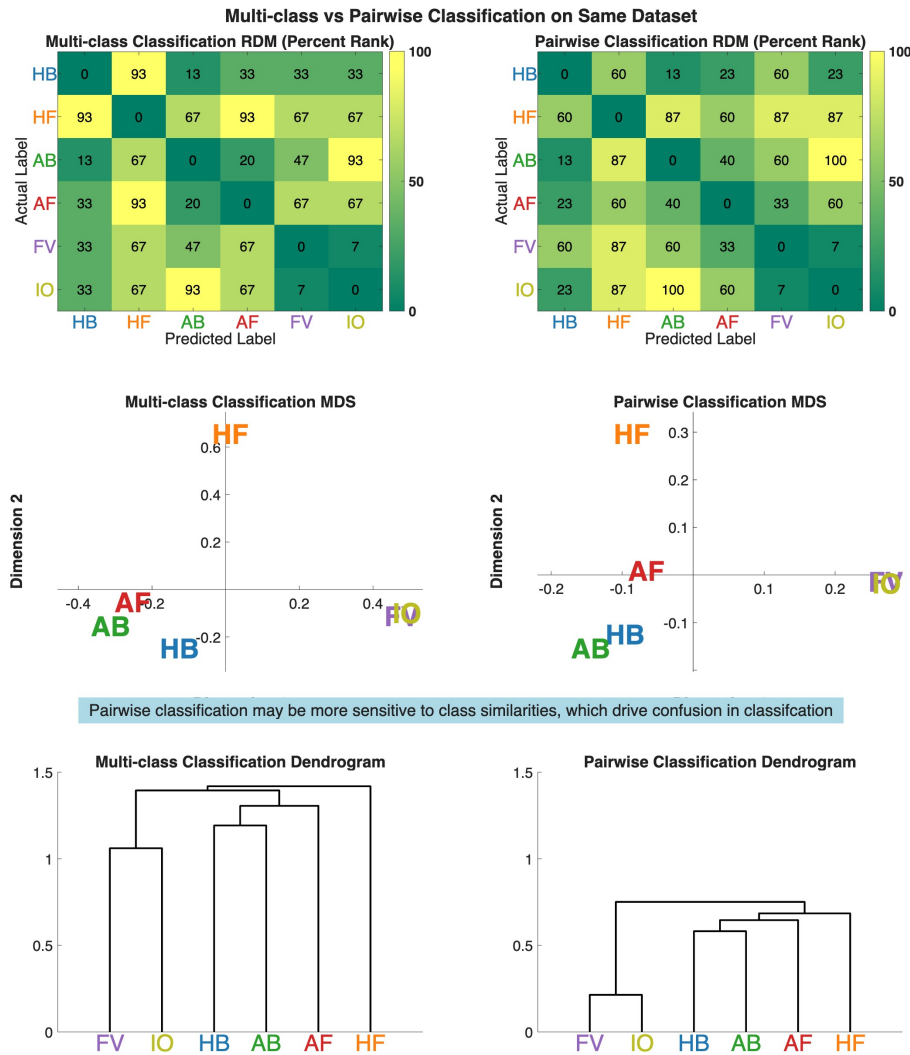


Figure 45: **Multiclass versus pairwise classification RDMs.** Data from participant S06 underwent a six-class classification using electrodes and timepoints. Left column: RDM, MDS, and dendrogram for multi-class confusion matrix. Right column: Equivalent visualizations for pairwise classification. The two classifications produce similar proximity spaces overall, with multiclass classifications resulting in larger distance values.

Comparison of non-classification RDMs. Next, we visualized two six-class non-classification RDMs—computed directly from the input data using Pearson correlation and Euclidean distance—as well as their corresponding MSTs. First, we compared the RDMs at a single sensor (electrode 96). This analysis used the following code:

```
singleElectrodeX = squeeze(X(96,:,:));
```

```
----- Compute RDM-----  
PearsonRDM = RDM_Computation.computePearsonRDM(singleElectrodeX, labels6, ...  
    'rngType', rnd_seed);  
  
EuclidRDM = RDM_Computation.computeEuclideanRDM(singleElectrodeX, labels6, ...  
    'rngType', rnd_seed);  
  
----- Plot -----  
Visualization.plotMatrix(PearsonRDM.RDM, ... 'rankType', 'p');  
Visualization.plotMatrix(EuclidRDM.RDM, ... 'rankType', 'p');
```

Results are shown in Figure 46. The percentile-ranked (for visualization only) RDMs are visually similar across the Pearson and Euclidean approaches. The MDS and MST plots, too, show a highly similar category structure across the two approaches, with differences in MDS coordinates and MST arrangement occurring mainly between the AB and AF categories relative to HB. The MDS and MST data scales differ between the two approaches, with the Pearson-based RDM showing larger MDS values yet smaller MST values than the Euclidean approach.

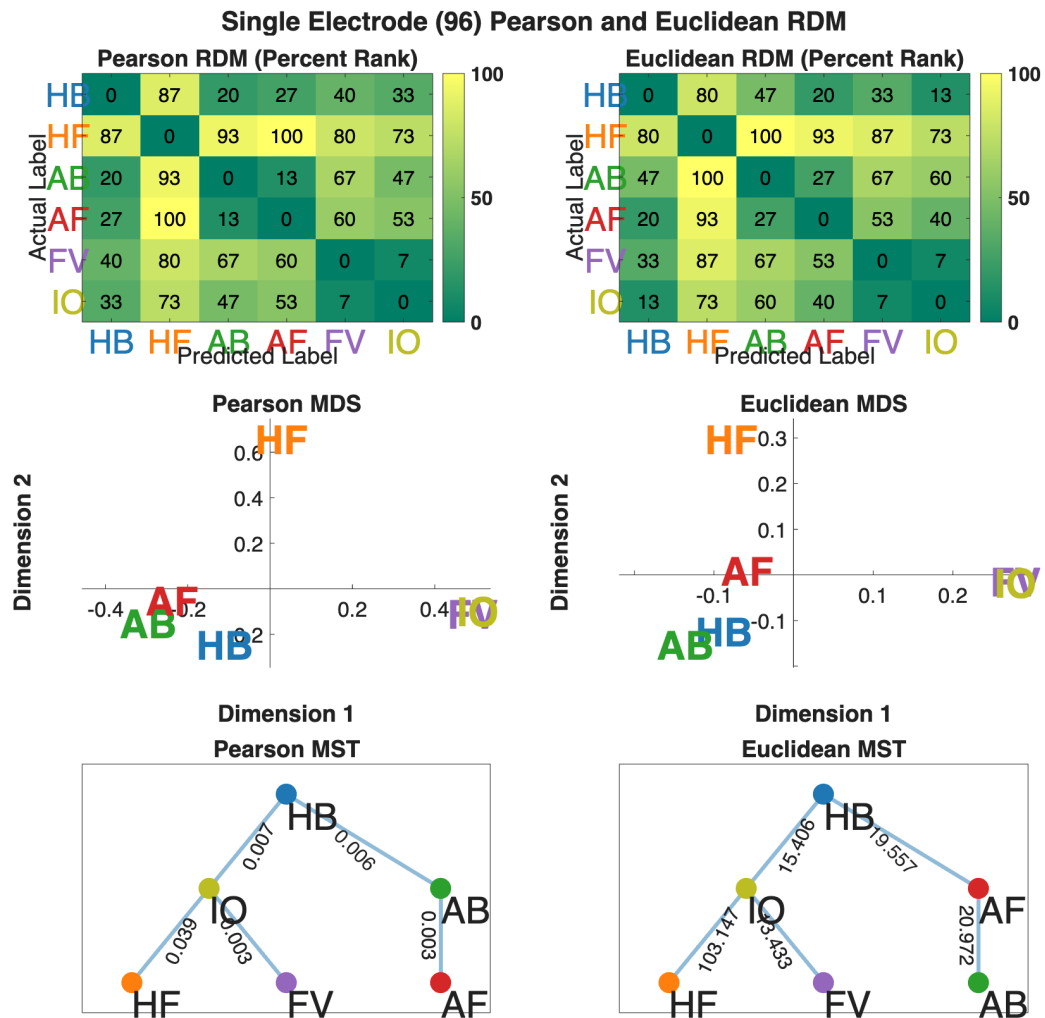


Figure 46: **Comparison of Pearson and Euclidean RDMs at electrode 96.** Each RDM type yields a distinct view of inter-class distances. Graph visualizations (bottom) show different MDS and MST structures that reflect the geometry implied by each metric.

Next, we compared the RDMs at a single timepoint (144 msec). For this we used the following code:

```
singleElectrodeX = squeeze(X(:,17,:));

----- Compute RDM-----
PearsonRDM = RDM_Computation.computePearsonRDM(singleElectrodeX, labels6, ...
    'rngType', rnd_seed);

EuclidRDM = RDM_Computation.computeEuclideanRDM(singleElectrodeX, labels6, ...
    'rngType', rnd_seed);

----- Plot -----
```

```
Visualization.plotMatrix(PearsonRDM.RDM, ... 'rankType', 'p');
Visualization.plotMatrix(EuclidRDM.RDM, ... 'rankType', 'p');
```

Results are shown in Figure 47. As in the previous figure, the percentile-ranked RDMs are quite similar across approaches, and the MDS and MST plots are broadly similar. As before, the Pearson RDM (without percentile ranking) corresponds to higher MDS coordinate values and smaller MST values overall than the Euclidean RDM.

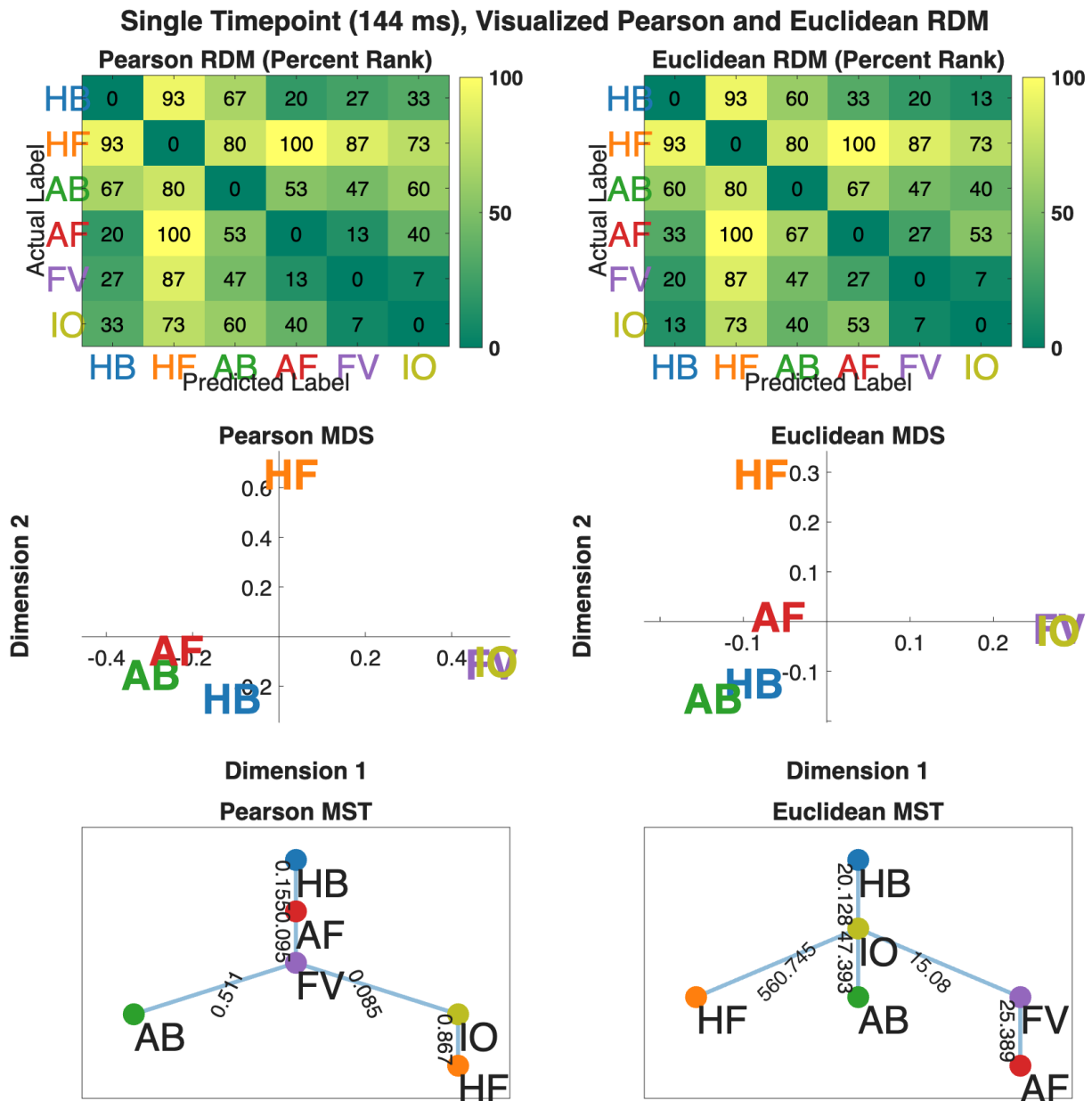


Figure 47: **Comparison of Pearson and Euclidean RDMs at timepoint 144 msec.** At 144 msec after stimulus onsets (when category-level ERPs are relatively well separated), both Pearson and Euclidean distances reflect structured category separation, with visible similarity structure preserved in MDS and MST representations.

Comparison of non-classification and classification RDMs. In our final analysis, we compared single-electrode non-classification RDM (based on Pearson correlation) with a classification RDM (based on multiclass LDA). This analysis used the following code:

```
singleElectrodeX = squeeze(X(96,:,:));

----- Preprocessing and Classssification -----
[xShuf, yShuf] = Preprocessing.shuffleData(X, labels72, 'rngType', rnd_seed);
xNorm = Preprocessing.noiseNormalization(xShuf, yShuf);
[xAvg, yAvg] = Preprocessing.averageTrials(xNorm, yShuf, 5, ...
    'handleRemainder', 'newGroup'...);

M = Classification.crossValidateMulti(xAvg, yAvg, 'PCA', 0.99);

----- Compute RDM -----
ClassRDM = RDM_Computation.computeCMRDM(M.CM);
PearsonRDM = RDM_Computation.computePearsonRDM(singleElectrodeX, labels72, ...
    'rngType', rnd_seed);

----- Plot -----
Visualization.plotMatrix(PearsonRDM.RDM, [...], 'rankType', 'p');
Visualization.plotMatrix(ClassRDM, [...], 'rankType', 'p');

Visualization.plotMDS(PearsonRDM.RDM, 'nodeColors', nodeColors);
Visualization.plotMDS(ClassRDM, 'nodeColors', nodeColors);
```

The RDMs and MSTs for this analysis are shown in Figure 48. In contrast to the 6-class comparisons, the 72-class results vary more between approaches. First, in the percentile-ranked RDMs, for the Pearson approach we see greater blocking in some areas of the matrix, reflecting different aspects of category structure (e.g., Human Face in rows 13 to 24 and Animal Body/Face in rows 25 to 48). This stronger category-level grouping is also evident in the MDS plot, where orange dots representing Human Face exemplars and red dots representing Animal Face exemplars appear to form especially cohesive category-level clusters. While color-based grouping of items appears to be more diffuse for the Classification RDM and MDS overall, Human Face exemplars are relatively distinguishable from the other categories.

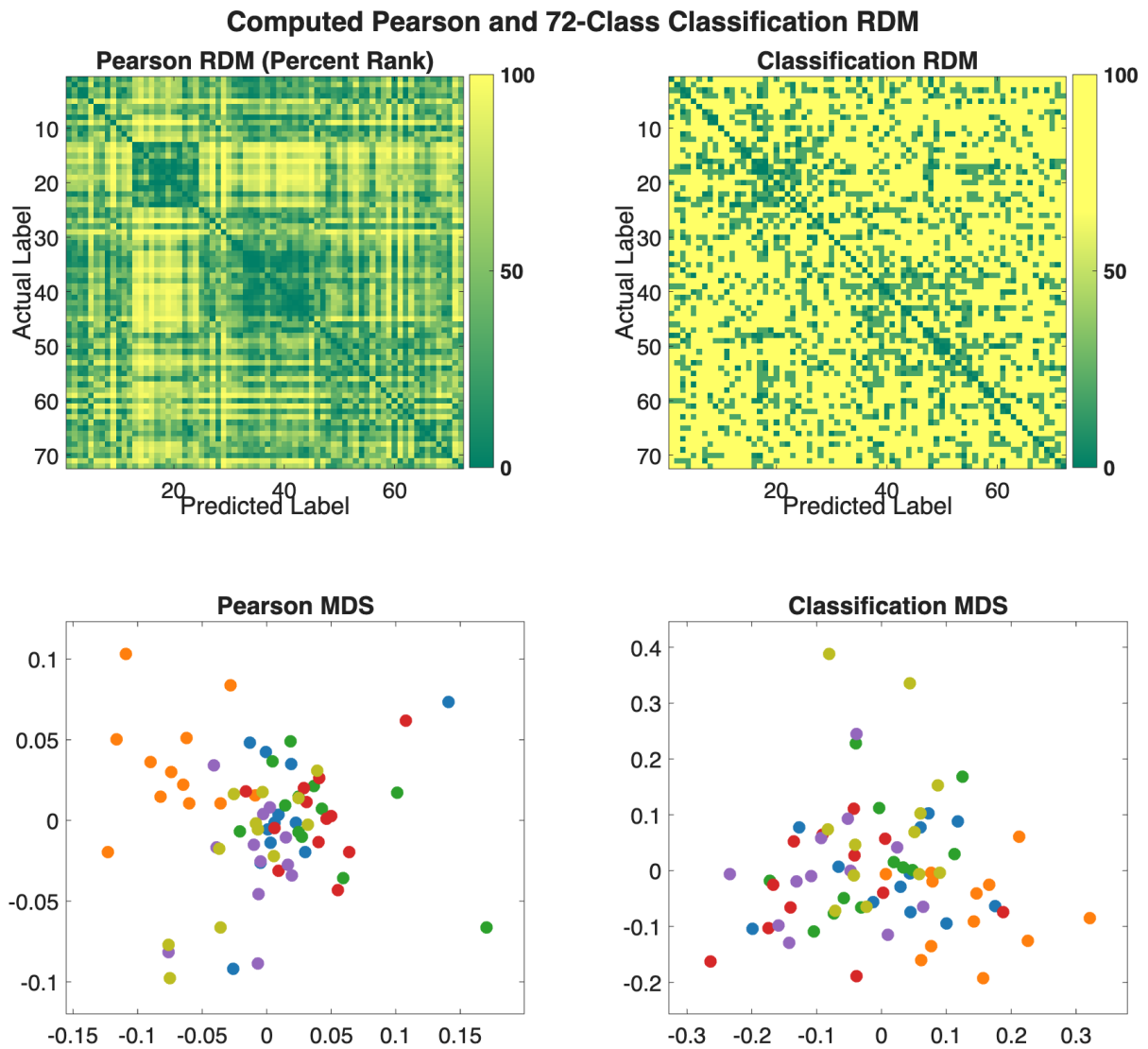


Figure 48: **Comparison of Pearson and Classification RDMs for 72 classes.** We constructed 72-class RDMs using channel 96 data from `S06.mat` and Pearson and multiclass classification. Compared to 6-class RDMs, we observe more variability across approaches for the 72-class case. The Pearson RDM shows greater blocking along various regions of the diagonal, which is then reflected in the MDS plot as stronger category grouping—for example, for Human Face (orange) and Animal Face (red) exemplars. In contrast, the classification approach shows weaker category structure in the RDM, and within-category exemplars are more loosely grouped, although Human Face (orange) exemplars still appear to separate from the other categories.

DISCUSSION

This analysis illustrates how the various approaches to constructing RDMs emphasize different properties of the data. Depending on the data and category groupings, the visualized RDMs and accompanying clustering visualizations may look similar or markedly different

across different RDM construction techniques. Users are thus encouraged to examine how the proximity space of their data vary according to which technique is used.

8.7 Illustrative 6: Customizing figures with stimulus images

`illustrative_6_figureCustomizations.m`

DESCRIPTION

While the current version of MatClassRSA does not include built-in support for adding stimulus images to figures, users can manually annotate figures with images of their choice. As an example, this final illustrative script demonstrates how to customize the `plotMatrix()` figure from the Visualization module with stimulus images.

Code and Data

This analysis uses the following toolbox functions in the following contexts:

- `Preprocessing.shuffleData()` — Randomly shuffles class labels. Used here prior to classification and averaging, to reduce temporal or order-based bias.
- `Preprocessing.noiseNormalization()` — Applies whitening to emphasize shared signal and reduce noise covariance structure, useful for within- and across-subject classification.
- `Preprocessing.averageTrials()` — Aggregates multiple trials into "pseudotrials" to reduce noise and increase feature strength.
- `Classification.crossValidateMulti()` — Performs within-subject classification using cross-validation. Returns accuracy and confusion matrices.
- `RDM_Computation.computeCMRDM()` — Computes representational dissimilarity matrices from confusion matrices.
- `Visualization.plotMatrix()` — Plots confusion matrices or representation dissimilarity matrices (RDMs) with optional group labeling and class-label color coding.

This analysis uses the following dataset:

- `S01.mat` — A three-dimensional EEG dataset (electrode \times time \times trial) from one participant. Included in the GitHub repository. Contains responses to 72 stimuli grouped into 6 classes.

RESULTS AND INTERPRETATION

Customize confusion matrix with category images as axis labels. We first perform the following function calls to obtain a confusion matrix:

```
[xShuf, yShuf] = Preprocessing.shuffleData(X, labels6, ...
    'rngType', rnd_seed);

xNorm = Preprocessing.noiseNormalization(xShuf, yShuf);

[xAvg, yAvg] = Preprocessing.averageTrials(xNorm, yShuf, 40, ...
    'handleRemainder', 'newGroup', 'rngType', rnd_seed);

M = Classification.crossValidateMulti(xAvg, yAvg, 'classifier', 'LDA', ...
    'PCA', 0.99);
```

Next, we add stimulus images as x and y Tick labels, in `plotMatrix()`, as follows:

```
Visualization.plotMatrix(M.CM, 'colorbar', 0, ...
    'colorMap', 'summer', 'matrixLabels', 1);
mainAx = gca;

hold(mainAx, 'on');
yl = get(mainAx, 'YLim');
xl = get(mainAx, 'XLim');

for i = 1:nImages
    yc = rowCenter(i);
    yBottom = yc - imgHeight/2;
    yTop = yc + imgHeight/2;
    image(mainAx, [xLeft, xLeft + imgWidth], [yBottom, yTop], ...
        stimImages{i}, 'Clipping', 'off');
end

for i = 1:nImages
    % match the column width
    xLeftCol = i - 0.4;
    xRightCol = i + 0.4;

    % below the matrix
    yBottom = bottomY - imgOffset - imgHeightX;
```

```

yTop      = bottomY - imgOffset;

image(mainAx, [xLeftCol, xRightCol], [yBottom, yTop], ...
      stimImages{i}, 'Clipping', 'off');
end

```

Results are shown in Figure 49. As can be seen in the figure, each of the six stimulus categories is now labeled with an image rather than with text.

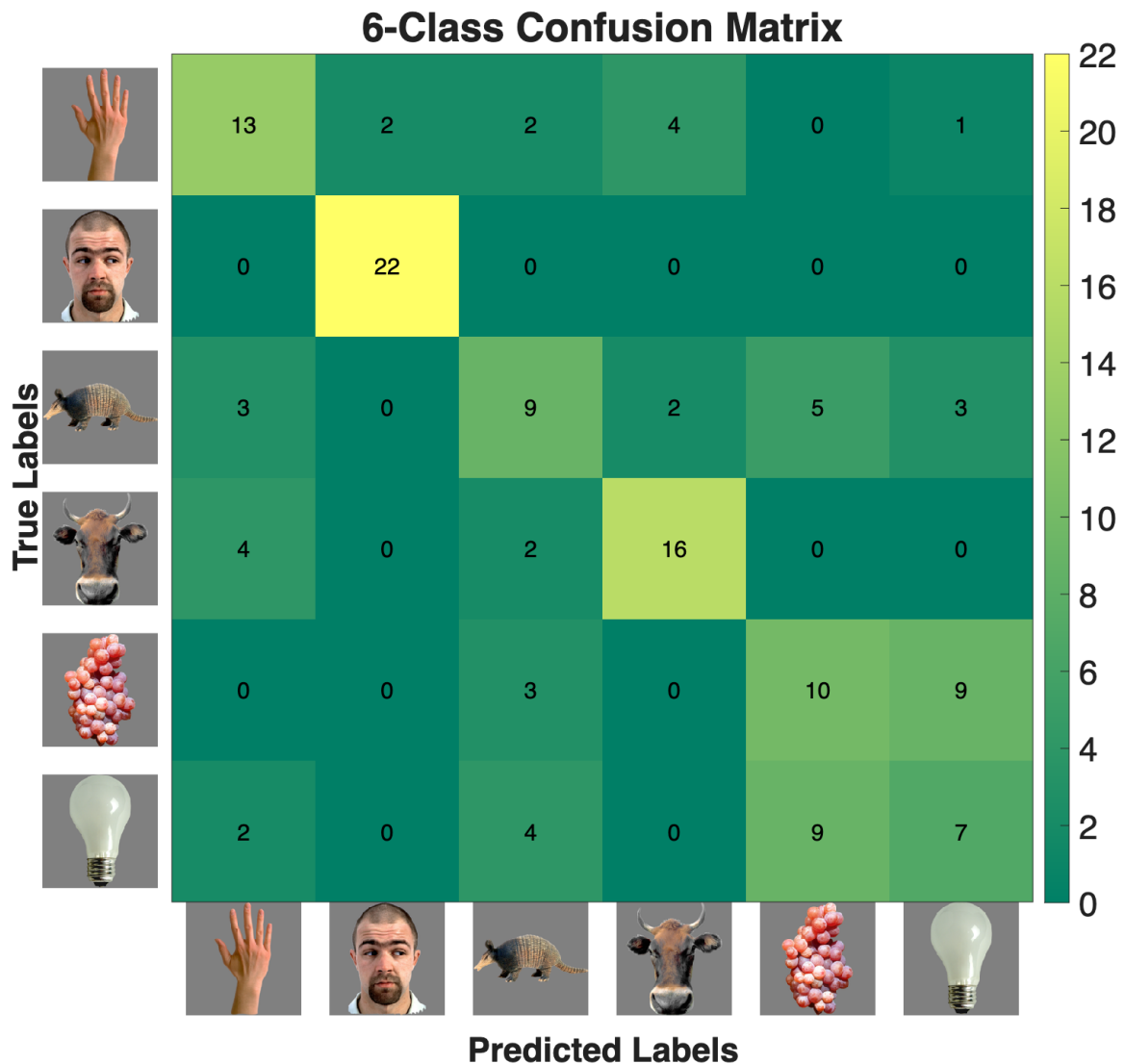


Figure 49: **Customization of 6-class confusion matrix with example images from each stimulus category.** Classification of subject S01, with visual stimuli used for evoked potential classification plotted as x-tick and y-tick labels.

DISCUSSION

Users can customize MatClassRSA visualization functions in order to achieve their desired plot aesthetics.

Appendix: Helper functions

This section contains brief descriptions of the MatClassRSA helper functions, which are all contained in the +Utils folder. Helper functions are documented in alphabetical order. For each helper function, we provide the syntax, a brief narrative description of its inputs and outputs, and its usage by other user-called and/or helper MatClassRSA functions.

centerAndScaleData()

Syntax

```
[xOut, centerOut, scaleOut] = centerAndScaleData(xIn, ...  
    centering, scaling);
```

Description. This function centers and scales each column of a 2D trial-by-feature data matrix. Its inputs are the 2D data matrix `xIn` (required) as well as optional centering and scaling specification inputs. The main output is the centered and/or scaled data matrix `xOut`. The function also outputs `centerOut` and `scaleOut`, which detail the centering and scaling parameters, respectively (e.g., when computing these values from training data and later applying to test data).

Usage. This function calls other helper functions `cube2trRows()` and `trRows2cube()`. It is called by the `predict()` and `trainMulti_opt()` functions from the Classification module as well as by the helper function `cvData()`.

`checkInputDataShape()`

Syntax

```
checkInputDataShape(X, Y);
```

Description. This function takes in a data matrix X and labels vector Y . It checks to ensure that X is a 2D or 3D matrix and ensures that the length of Y matches the size of the trial dimension of X . Finally, it transposes Y to a column if it is not already a column vector. The function has no outputs but rather prints an error message if any of the above conditions are not met.

Usage. This function does not call any other helper functions. It is called by the `trainMulti()` and `trainMulti_opt()` functions from the Classification module.

`classTuple2Nchoose2Ind()`

Syntax

```
y = classTuple2Nchoose2Ind(classTuple, n);
```

Description. Given $\binom{N}{2}$ (i.e., N choose 2) pairs of classes which are sequentially ordered in a vector of tuples, this function takes in a pair of classes (the input `classTuple`) as well as the total number of classes n and returns the index of the input pair in the vector of pairs. For example, if conducting pairwise classification amongst 5 classes, we would have $\binom{5}{2}$ tuples of classes. If we were to call this function as `classTuple2Nchoose2Ind((2, 4), 5)`, the function would return index 6.

Usage. This function does not call any other helper functions. It is called by the `crossValidatePairs_opt()` function from the Classification module and by the helper function `decValues2PairwiseAcc()`.

`colors.mat`

Syntax

```
load(['Utils' filesep 'colors.mat']);  
load(['Utils' filesep 'colors.mat'], 'rgb10');
```

Description. This is a `.mat` data file containing specifications for a 10-color palette, 20-color palette (original and light versions of the 10-color palette), and others. The 10-color palette is used in MatClassRSA illustrative analyses, and has been used in previous classification papers by members of our author team (Kaneshiro et al., 2015b; Kong et al., 2020; Losorelli et al., 2020). The 10- and 20-color palettes are illustrated in Figure 50. The palette specifications originated from the d3 GitHub repo (specific page no longer available).³⁶

³⁶<https://github.com/d3/d3/wiki/Ordinal-Scales>

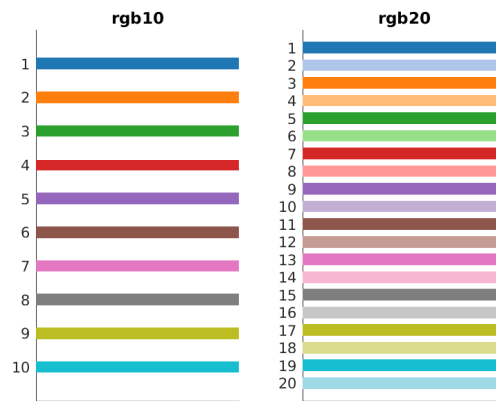


Figure 50: 10- and 20-color palettes from `colors.mat`.

Usage. This data file cannot call any functions and is not called by any of the core MatClassRSA functions. The `rgb6` variable separately specified within several of the MatClassRSA illustrative analysis scripts represents a subset of the `rgb10` palette provided in `colors.mat`.

`computeAccuracy()`

Syntax

```
acc = computeAccuracy(actualY, predictedY);
```

Description. This function takes in the actual labels `actualY` and predicted labels `predictedY` of a classification and returns the percent accuracy of the classification `acc` as a value between 0 and 1 (i.e., the proportion of predictions for which the actual label and predicted label were the same).

Usage. This function does not call any other helper functions. It is called by the `crossValidateMulti()`, `crossValidateMulti_opt()`, `crossValidatePairs()`, and `predict()` functions from the Classification module as well as the helper function `nestedCvGridSearch()`.

`computeReliability()`

Syntax

```
rels = computeReliability(data, labels, num_permutations);
```

Description. This function takes in a 2D space-by-trial matrix `data` relating to a specific time point; a labels vector `labels`; and a specification of split-half permutations `num_permutations` and computes the space-wise reliability of the input data. The output

rels is of size nPermutations by nComponents (space) and thus provides the reliability for each electrode at the provided time point.

Usage. This function does not call any other helper functions. It is called by the computeSampleSizeReliability() and computeSpaceTimeReliability() functions from the Reliability module.

convert2double()

Syntax

```
[X, Y] = convert2double(X, Y);
```

Description. This function takes in a data matrix X and labels vector Y, converts each one to double format if not already double, and returns the double versions X and Y.

Usage. This function does not call any other helper functions. It is called by the crossValidateMulti(), trainPairs(), and trainPairs_opt() functions from the Classification module.

convertSimToDist()

Syntax

```
[normData, sigmaInv] = Utils.noiseNormalization(X, Y);
```

Description. This function converts a similarity matrix to a distance matrix. The first input xIn is a square e.g., confusion matrix. The second input distType specifies the type of distance (linear, power, logarithmic, pairsie, or none). The third input distPower is the distance power used in power and logarithmic computations. All three inputs must be provided. The function returns the resulting distance matrix xOut.

Usage. This function does not call any other helper functions. It is called by the computeCMRDM() function from the RDM_Computation module.

cov1para()

Syntax

```
[sigma, shrinkage] = cov1para(x, shrink);
```

Description. This function applies shrinkage to compute a smoothed estimate of the covariance matrix. It takes in space-by-trial matrix x and optional shrinkage parameter shrink. It outputs the invertible covariance estimator sigma as well as the specified or computed shrinkage parameter shrinkage.

This function was created by Olivier Ledoit and Michael Wolf and was obtained from the tutorial code provided by Guggenmos et al. (2018).³⁷

Usage. This function does not call any other helper functions. It is called by the `noiseNormalization()` function from the Preprocessing module.

cube2trRows()

Syntax

```
xOut = cube2trRows(xIn);
```

Description. This function takes in a 3D space-by-time-by-trials matrix `xIn` and reshapes it to a 2D matrix `xOut` in which each row is a trial and the columns are the concatenated time vectors from each electrode (i.e., data from electrode 1 followed by data from electrode 2, etc.).

Usage. This function does not call any other helper functions. It is called by the `averageTrials()` function from the Preprocessing module. It is also used in the following other helper functions: `centerAndScaleData()`, `subsetTrainTestMatrices()`, and `trRows2cube()`.

cvData()

Syntax

```
[obj, V, nPC, colMeans, colScales] = cvData(X, Y, trainDevTestSplit, ...  
      ip, center, scale, nFolds);
```

Description. `cvData` is an object that stores data to be used for cross validation. It takes as input the data matrix `X`, labels vector `Y`, `trainDevTestSplit` object, and PCA parameters specified in the `classifyCrossValidate()` function call. It formats the data into partitions to enable convenient cross validation later.

Usage. This function calls helper functions `centerAndScaleData()`, `cube2trRows()`, and `getPCs()`. It is called by the following functions in the Classification module: `crossValidateMulti()`, `crossValidateMulti_opt()`, `crossValidatePairs()`, `crossValidatePairs_opt()`, `predict()`, `trainMulti()`, `trainMulti_opt()`, `trainPairs()`, and `trainPairs_opt()`. It is also called by the helper functions `nestedCvGridSearch()` and `trainDevTestPart()`.

decValues2PairwiseAcc()

Syntax

³⁷<https://github.com/m-guggenmos/megmvpa/tree/master>

```
[pairwiseAccuracies, pairwiseCMs, pairwiseCell] = ...  
    decValues2PairwiseAcc(pairwiseCMs, testY, labels, decVals, ...  
    pairwiseCell);
```

Description. When conducting pairwise classification using LIBSVM, this function is used to convert the decision values output from LIBSVM to pairwise accuracies. It takes in `pairwiseCMs`, a 3D matrix to store all pairwise CMs; labels vector `testY`; predicted labels vector `labels`; SVM decision values `decVals`; and pairwise classification output `pairwiseCell`. The function outputs pairwise accuracies `pairwiseAccuracies`, updated pairwise CMs `pairwiseCMs`, and updated classification output `pairwiseCell`.

Usage. This function calls other helper functions `classTuple2Nchoose2Ind()` and `getNChoose2Ind()`. It is called by the `crossValidatePairs_opt()` and `predict()` functions from the Classification module, and by the helper function `getNChoose2Ind()`.

`fitModel()`

Syntax

```
[mdl, scale] = fitModel(X, Y, ip, gamma, C);
```

Description. Given the classifier and training data specified in another function, this function returns a classification model. Specifically, the inputs are 2D trial-by-feature training data matrix `X`, labels vector `Y`, classifier input parser parameters `ip`, and (for SVM classifications) hyperparameters `gamma` and `C`. The function outputs an object `mdl` which contains the classification model, as well as `scale`, which contains scale and shift parameters for SVM classifications.

Usage. This function calls the helper function `scaleDataInRange()`. It is called by the `crossValidateMulti()`, `crossValidatePairs()`, `crossValidatePairs_opt()`, `predict()`, `trainMulti()`, and `trainMulti_opt()` functions from the Classification module. It is also called by helper functions `modelPredict()`, `nestedCvGridSearch()`, and `permuteModel()`.

`getNChoose2Ind()`

Syntax

```
[firstInds, secondInds] = getNChoose2Ind(n);
```

Description. Given an input length `n`, this function returns two arrays representing the first and second classes for e.g., pairwise classifications of `n` classes.

Usage. This function does not call any other helper functions. It is called by the helper function `decValues2PairwiseAcc()`.

getPCs()

Syntax

```
[y, V, nPC] = getPCs(X, PCs);
```

Description. This function computes principal components via singular value decomposition (SVD). It takes in the data matrix *X* and PCA specification *PCs*. If *PCs* is a positive integer, it specifies the number of PCs to use. If *PCs* is a value between 0 and 1, it specifies that the number of PCs used should be as many as specifies that proportion of the variance. The function outputs matrix *Y*, SVD parameter *V*, and number of PCs extracted *nPC*.

Usage. This function does not call any other helper functions. It is called by the `predict()` and `trainMulti_opt()` functions from the Classification module, as well as by the helper function `cvData()`.

getTickCoord()

Syntax

```
[xTickVec yTickVec] = getTickCoord();
```

Description. This function returns the x and y coordinates of the ticks on the plot. It has no inputs or outputs.

Usage. This function does not call any other helper functions. It is called by the `plotDendrogram()` and `plotMatrix()` functions from the Visualization module.

initInputParser()

Syntax

```
[normData, sigmaInv] = Utils.noiseNormalization(X, Y);
```

Description. This function initializes the input parser for various functions. It fills in generalized parameters such as `rng`, PCA specifications, center and scale parameters, data subsetting, and classification parameters. It also fills in additional parameters depending on which function it is being called from. It takes in the calling function `functionName`, initialized input parser `ip`, data matrix *X*, labels vector *Y*, and other optional arguments. It outputs *y*, an updated version of the input parser.

Usage. This function calls the helper function `is2Dor3DMatrix()`. It is called by the following functions in the Classification module: `crossValidateMulti()`, `crossValidateMulti_opt()`, `crossValidatePairs()`, `crossValidatePairs_opt()`, `trainMulti()`, `trainMulti_opt()`, `trainPairs()`, and `trainPairs_opt()`.

`initPairwiseCellMat()`

Syntax

```
pairwiseCellMat = initPairwiseCellMat(numClasses);
```

Description. For specified number of classes `numClasses`, this function initializes a `numClasses`-by-`numClasses` cell array called `pairwiseCellMat`. Each off-diagonal element of `pairwiseCellMat` is a struct with fields for the confusion matrix (CM — 2 x 2 matrix initialized as all zeros), the pair label (`classBoundary` — e.g., '1 vs. 2'), and the accuracy (`accuracy` — initialized as NaN).

Usage. This function does not call any other helper functions. It is called by the following functions of the Classification module: `crossValidatePairs()`, `crossValidatePairs_opt()`, `trainPairs()`, `trainPairs_opt()`, and `predict()`. It is also used by helper function `decValues2PairwiseAcc()`.

`is2Dor3DMatrix()`

Syntax

```
y = is2Dor3DMatrix(x);
```

Description. This function returns output `y` as 1 if the input `x` is a 2D (including vector) or 3D matrix.

Usage. This function does not call any other helper functions. It is called by the `predict()` function from the Classification module and by the helper function `initInputParser()`.

`locsEGI124.mat`

Syntax

```
load(['Utils' filesep 'locsEGI124.mat']); % Loads 'locs' variable  
Utils.topoplotStandalone(Values, locs); % 'locs' variable is 2nd input
```

Description. This is a `.mat` data file containing sensor location coordinates for the 124-channel data used in the illustrative analyses (see Chapter 8.3). Specifically, the coordinates reflect electrodes 1–124 of the 128-channel net³⁸ of the Electrical Geodesics, Inc.³⁹ system.

Usage. This data file cannot call any functions and is not called by any of the core MatClassRSA functions. It is called by the `topoplotStandalone()` helper function as part of Illustrative Analysis 2 (single-channel analyses; Chapter 8.3). **Note:** For MatClassRSA examples specifically, the `topoplotStandalone()` function takes in the `locs` *variable* loaded from `locsEGI124.mat` due to difficulties in inputting filenames of sensor-location maps to

³⁸<http://bit.ly/4nN2wIB>

³⁹<https://www.egi.com/>

that function when such a map file is stored in the +Utils folder. However, in general use, the *filename* of any typical sensor-location file (e.g., .loc, .sfp format⁴⁰) in the user's path can be specified as the second input to the `topoplotStandalone()` function.

modelPredict()

Syntax

```
[predictions, decision_values] = modelPredict(X, mdl, scale);
```

Description. This function takes in a 2D trial-by-feature test data matrix `X`, classification model `mdl` (output from `fitModel()`), and `scale` struct (for SVM classifications only) and predicts the labels of the input data. It outputs predicted labels in `predictions` and, for SVM classifications, the decision values in `decision_values`.

Usage. This functions calls the helper functions `scaleDataShiftDivide()` and `SVMhandleties()`. It is called by the `crossValidateMulti()`, `crossValidatePairs()`, `crossValidatePairs_opt()`, and `predict()` functions from the Classification module as well as by helper functions `nestedCvGridSearch()` and `permuteModel()`.

nestedCvGridSearch()

Syntax

```
[gamma_opt, C_opt] = nestedCvGridSearch(X, Y, ip, cvDataObj, ...  
    excludeIndx);
```

Description. Given training data matrix `X`, labels vector `Y`, and a vector of gammas and `Cs` to search over, this function runs over a grid of all possible combinations of gammas and `Cs` to find the values that produce the highest cross validation accuracy. Gamma is a hyperparameter of the rbf kernel for SVM classification, and `C` is a hyperparameter of both the rbf and linear kernels for SVM classification.

Usage. This function calls the helper functions `computeAccuracy()`, `fitModel()`, and `modelPredict()`. It is called by the `crossValidateMulti_opt()`, `crossValidatePairs_opt()`, `trainMulti_opt()`, and `predict()` functions from the Classification module.

normalizeMatrix()

Syntax

```
xOut = normalizeMatrix(xIn, normType);
```

⁴⁰<https://eeglab.org/tutorials/ConceptsGuide/coordinateSystem.html>

Description. This function normalizes the input matrix `xIn` as specified by `normType` (divide by diagonal or sum; no normalization; or subtract 0.5 and then divide by 0.5).

Usage. This function does not call any other helper functions. It is called by the `computeCMRDM()` function from the `RDM_Computation` module.

`permTestPVal()`

Syntax

```
p = permTestPVal(value, permVector, [direction]);
```

Description. This function takes in an observed value of interest `value`, a vector of other values `permVector` (presumably the null distribution from permutation testing), and optionally the direction of the test `direction`, and computes the percentile of the observed value among the null distribution. If `direction` is not specified it will default to upper-tail calculation. This function was informed from a MATLAB Answers post.⁴¹

Usage. This function does not call any other helper functions. It is called by the `crossValidateMulti()`, `crossValidateMulti_opt()`, `crossValidatePairs()`, `crossValidatePairs_opt()`, and `predict()` functions from the `Classification` module.

`processRDM()`

Syntax

```
[normData, sigmaInv] = Utils.noiseNormalization(X, Y);
```

Description. For a given input RDM, this function ensures the RDM is square; ensures the RDM is symmetric (if not, will print a warning and use the lower triangle); and ensures the diagonal is zero (if not, will print a warning and set diagonal to zero). The function output `y` will be the RDM with these three conditions enforced.

Usage. This function does not call any other helper functions. It is called by the `plotDendrogram()`, `plotMDS()`, and `plotMST()` functions from the `Visualization` module.

`processTrainDevTestSplit()`

Syntax

```
[normData, sigmaInv] = Utils.noiseNormalization(X, Y);
```

Description. This function checks that the train/development/test data splits used for optimization functions abide by the restrictions of the data and cross validation procedure.

⁴¹<https://bit.ly/3FVdryZ>, accessed April 5, 2025.

If array was in decimal format, it then converts the decimals representing fractions to integers representing number of trials in each train/development/test fold. The inputs are `trainDevTestSplit` — which can be a 2- or 3- element vector depending on whether the split is train/test or train/development/test, respectively — and full training data matrix `X`. The output is `y`, the updated version of the `trainDevTestSplit` input.

Usage. This function does not call any other helper functions. It is called by the `trainMulti()` and `trainMulti_opt()` functions from the Classification module and by helper function `trainDevTestPart()`.

`rankDistances()`

Syntax

```
xOut = rankDistances(xIn, rankType);
```

Description. This function takes in a symmetric distance matrix `xIn` and ranks the distances as specified by `rankType` (rank, percent rank, or no rank). It returns ranked matrix `xOut`.

Usage. This function does not call any other helper functions. It is called by the `computeCMRDM()` function from the RDM_Computation module and the `plotMatrix()` function from the Visualization module.

`scaleDataInRange()`

Syntax

```
[xScaled, shift1, shift2, scaleFactor] = scaleDataInRange(xIn,  
    desiredMinMax);
```

Description. This function takes in the data matrix `xIn` plus an optional vector of min and max values `desiredMinMax`, and scales its values so that its values fall between the specified output min and max. If the output min-max vector is not specified or is empty, the function will scale the data to the range [0, 1]. Scaling is based upon the min and max values of the entire input data matrix.

Usage. This function does not call any other helper functions. It is called by helper functions `fitModel()` and `scaleDataShiftDivide()`.

`scaleDataShiftDivide()`

Syntax

```
scaledData = scaleDataShiftDivide(data, shift1, shift2, scaleFactor);
```

Description. This function takes in the data matrix data along with shift and divide factors calculated from the function `scaleDataInRange()`. The function applies the shift and divide factor on the data matrix to bring the data to the range specified in `scaleDataInRange()`. Thus, the function can be used to apply precomputed shift and scaled factors to a dataset.

Usage. This function does not call any other helper functions. It is called by the helper function `modelPredict()`.

setUserSpecifiedRng()

Syntax

```
setUserSpecifiedRng(r);
```

Description. This function sets the rng specification as indicated by input `r`. The function does not have any outputs, but sets the `rngType` for the calling function with appropriate input specifications as outlined in the documentation for those functions:

- **rngType — Random number generator (rng) specification.** If `rngType` is not entered or is empty, `rng` will be assigned here as `{'shuffle', 'twister'}`. The `rngType` input can be specified in the following ways:
 - Single acceptable `rng` specification input (e.g., `1`, `'default'`, `'shuffle'`); in these cases, the generator will be set to `'twister'`.
 - Dual-argument specifications as either a 2-element cell array (e.g., `{'shuffle', 'twister'}`) or string array (e.g., `["shuffle", "twister"]`).
 - `rng` struct as previously assigned by `rngType = rng`.

Usage. This function does not call any other helper functions. It is called by the following functions:

- Reliability module: `computeSampleSizeReliability()`, `computeSpaceTimeReliability()`
- Preprocessing module: `averageTrials()`, `shuffleData()`
- Classification module: `crossValidateMulti()`, `crossValidateMulti_opt()`, `crossValidatePairs()`, `crossValidatePairs_opt()`, `trainMulti()`, `trainMulti_opt()`, `predict()`
- RDM_Computation module: `computeEuclideanRDM()`, `computePearsonRDM()`

subsetTrainTestMatrices()

Syntax

```
[W, nSpace, nFeature, nTrials] = subsetTrainTestMatrices(X, spaceUse,  
    timeUse, featureUse);
```

Description. This function subsets the input Matrix *X* according to parameters *spaceUse*, *timeUse* and *featureUse*, defined by the user during the function call. *spaceUse* and *timeUse* subset 3D space-by-time-by-trial input matrix *X* along dimensions 1 and/or 2, respectively, while *featureUse* subsets a 2D trial-by-feature input matrix *X* along dimension 2. The function returns a 2D trial-by-feature matrix *W* (whether a 2D or 3D matrix was input) along with three other outputs *nSpace* (for 3D input matrices), *nFeature* (for 2D input matrices, denotes length of time dimension for 3D input matrices), and *nTrials*.

Usage. This function calls the helper function `cube2trRows()`. It is called by all functions in the Classification module: `crossValidateMulti()`, `crossValidateMulti_opt()`, `crossValidatePairs()`, `crossValidatePairs_opt()`, `trainMulti()`, `trainMulti_opt()`, `trainPairs()`, `trainPairs_opt()`, and `predict()`.

SVMhandleties()

Syntax

```
[winnerIndex, tallies, tieFlag] = SVMhandleties(dec_vals, labels);
```

Description. This function is an auxiliary function to LIBSVM to handle ties. LIBSVM's multiclass classification is implemented using one-to-one classification. However, in the event that a trial's most classified class is tied by two or more different classes, this function is used to break the tie. This function is important because otherwise, during a multiclass tie, LIBSVM would default to the first class amongst the ties, inducing a bias in classification. This function takes in LIBSVM decision values *dec_vals* and labels *labels* and outputs the index of the winning label *winnerIndex*, number of votes *tallies*, and *tieFlag*, a debugging flag indicating if a tie was detected.

Usage. This function does not call any other helper functions. It is called by the helper function `modelPredict()`.

symmetrizeMatrix()

Syntax

```
xOut = symmetrizeMatrix(xIn, symmType);
```

Description. This function symmetrizes an input matrix `xIn` by operating on the matrix and its transpose according to parameters specified in `symmType` (arithmetic, geometric, or harmonic mean; or none).

Usage. This function does not call any other helper functions. It is called by the `computeCMRDM()` function from the `RDM_Computation` module.

`topoplotStandalone()`

Syntax

```
[handle, Zi, grid, Xi, Yi] = topoplot_new(Values, loc_file);
```

Description. This function plots a set of values (specified by `Values`) at the locations specified by `loc_file` as a 2D topographic map. The map is rendered from the perspective of looking down at the head, with the nose oriented to the top of the plot. It is a self-contained version of EEGLAB's `topoplot()`⁴² function and requires no EEGLAB dependencies in order to run.

This function was developed by the Parra Lab⁴³ and was obtained from a public GitHub repository related to Dmochowski et al. (2018).⁴⁴

Usage. This function does not call any other helper functions. It is called in Illustrative Analysis 2 (single-channel analyses; Chapter 8.3), and in that context makes use of the `locsEGI124.mat` data file also provided in the `+Utils` folder. **Note:** For MatClassRSA examples specifically, the `topoplotStandalone()` function takes in the `locs` *variable* loaded from `locsEGI124.mat` due to difficulties in inputting filenames of sensor-location maps to that function when such a map file is stored in the `+Utils` folder. However, in general use, the *filename* of any typical sensor-location file (e.g., `.loc`, `.sfp` format⁴⁵) in the user's path can be specified as the second input to the `topoplotStandalone()` function.

`trainDevGridSearch()`

Syntax

```
[gamma_opt, C_opt] = trainDevGridSearch(trainX, trainY, devX, devY, ip);
```

Description. Given training and development partitions of data (`trainX`, `devX`) and labels (`trainY`, `devY`), as well as an input parser containing hyperparameter `gamma` and `C` grid specifications (`ip`), this function runs cross validations over the grid of all possible combinations of `gamma` and `C` and returns the optimal value for each (`gamma_opt`, `C_opt`).

⁴²<https://scn.ucsd.edu/~arno/eeglab/auto/topoplot.html>

⁴³<https://parralab.org/>

⁴⁴https://github.com/dmochow/SRC/blob/master/topoplot_new.m

⁴⁵<https://eeglab.org/tutorials/ConceptsGuide/coordinateSystem.html>

Usage. This function does not call any other helper functions. It is called by the `crossValidateMulti_opt()`, `crossValidatePairs_opt()`, `trainMulti_opt()`, and `predict()` functions from the Classification module as well as by helper function `permuteModel()`.

trainDevTestPart()

Syntax

```
obj = trainDevTestPart(X, nFolds, trainDevTestSplit);
```

Description. This class stores a cross-validation partition. This constructor of this class takes in an input data matrix `X`, number of folds `nFolds`, and split specification `trainDevTestSplit`. The output `obj` specifies the data partitions. NOTE: This class is an alternative to the Matlab `cvpartition` class. The reason this class is used instead of the Matlab `cvpartition` class is because the Matlab class uses randomization to assign partitions. But in MatClassRSA, data shuffling is optionally done in the Preprocessing module (`shuffleData()`) and in the present step we choose to assign partitions sequentially.

Usage. This function calls the helper functions `cube2trRows()` and `processTrainDevTestSplit()`. It is called by the `crossValidateMulti()`, `crossValidateMulti_opt()`, `crossValidatePairs()`, `crossValidatePairs_opt()`, `trainMulti()`, and `trainMulti_opt()` functions from the Classification module.

trRows2cube()

Syntax

```
xOut = trRows2cube(xIn, N);
```

Description. This function takes in a 2D matrix trial-by-feature matrix `xIn`, where the feature dimension is assumed to represent data from concatenated channels (all data from electrode 1 followed by all data from electrode 2, etc.) as well as input `N` which specifies how many time samples are in each trial. The function reshapes the data to output matrix `xOut`, which is a 3D space-by-time-by-trials matrix.

Usage. This function does not call any other helper functions. It is called by the `averageTrials()` function from the Preprocessing module as well as by helper function `centerAndScaleData()`.

verifySVMParameters()

Syntax

```
verifySVMParameters(ip);
```

Description. This function is used for the non-optimization classification functions to ensure that gamma and C parameters are manually set by the user when using the SVM classifier. It takes in the input parser `ip` of the classification function calling this function. There are no outputs; If the gamma and C parameters are not appropriately set, the function will return an error with instructions to use one of the optimization functions to compute suitable values.

Usage. This function does not call any other helper functions. It is called by the `crossValidateMulti()`, `trainMulti()`, and `trainPairs()` functions from the Classification module.

References

- [1] Brendan Z Allison, Dennis J McFarland, Gerwin Schalk, Shi Dong Zheng, Melody Moore Jackson, and Jonathan R Wolpaw. “Towards an independent brain–computer interface using steady state visual evoked potentials”. In: *Clinical Neurophysiology* 119.2 (2008), pp. 399–408. DOI: <http://doi.org/10.1016/j.clinph.2007.09.121>.
- [2] Winko W An, Abigail Noyce, Alexander Pei, and Barbara Shinn-Cunningham. “Neural representation of spatial and non-spatial auditory attention in EEG signals”. In: *bioRxiv* (2023), pp. 2023–07.
- [3] Aurélien Appriou, Léa Pillette, David Trocellier, Dan Dutartre, Andrzej Cichocki, and Fabien Lotte. “BioPyC, an open-source python toolbox for offline electroencephalographic and physiological signals classification”. In: *Sensors* 21.17 (2021), p. 5740.
- [4] Benjamin Blankertz, Gabriel Curio, and Klaus-Robert Müller. “Classifying Single Trial EEG: Towards Brain Computer Interfacing”. In: *Advances in Neural Information Processing Systems*. 2002, pp. 157–164.
- [5] Benjamin Blankertz, Steven Lemm, Matthias Treder, Stefan Haufe, and Klaus-Robert Müller. “Single-trial analysis and classification of ERP components—a tutorial”. In: *NeuroImage* 56.2 (2011), pp. 814–825.

- [6] Thomas Carlson, David A. Tovar, Arjen Alink, and Nikolaus Kriegeskorte. “Representational dynamics of object vision: The first 1000 ms”. In: *Journal of Vision* 13.10 (2013). DOI: [10.1167/13.10.1](https://doi.org/10.1167/13.10.1). eprint: <http://www.journalofvision.org/content/13/10/1.full.pdf+html>.
- [7] David Cheriton and Robert Endre Tarjan. “Finding minimum spanning trees”. In: *SIAM journal on computing* 5.4 (1976), pp. 724–742.
- [8] Radoslaw Martin Cichy, Dimitrios Pantazis, and Aude Oliva. “Resolving human object recognition in space and time”. In: *Nature neuroscience* 17.3 (2014), pp. 455–462.
- [9] Arnaud Delorme and Scott Makeig. “EEGLAB: An open source toolbox for analysis of single-trial EEG dynamics including independent component analysis”. In: *Journal of Neuroscience Methods* 134.1 (2004), pp. 9–21. DOI: <http://dx.doi.org/10.1016/j.jneumeth.2003.10.009>.
- [10] Jacek P Dmochowski, Jason J Ki, Paul DeGuzman, Paul Sajda, and Lucas C Parra. “Extracting multidimensional stimulus-response correlations using hybrid encoding-decoding of neural activity”. In: *NeuroImage* 180.Pt A (2018), pp. 134–146. DOI: [10.1016/j.neuroimage.2017.05.037](https://doi.org/10.1016/j.neuroimage.2017.05.037).
- [11] Richard Dubes and A K Jain. “Clustering methodologies in exploratory data analysis”. In: *Advances in Computers*. Ed. by Marshall C Yovits. Vol. 19. Elsevier, 1980, pp. 113–228. DOI: [http://dx.doi.org/10.1016/S0065-2458\(08\)60034-0](http://dx.doi.org/10.1016/S0065-2458(08)60034-0).
- [12] Johannes J Fahrenfort, Joram Van Driel, Simon Van Gaal, and Christian NL Olivers. “From ERPs to MVPA using the Amsterdam decoding and modeling toolbox (ADAM)”. In: *Frontiers in neuroscience* 12 (2018), p. 368.
- [13] Lawrence Ashley Farwell and Emanuel Donchin. “Talking off the top of your head: toward a mental prosthesis utilizing event-related brain potentials”. In: *Electroencephalography and Clinical Neurophysiology* 70.6 (1988), pp. 510–523. DOI: [http://doi.org/10.1016/0013-4694\(88\)90149-6](http://doi.org/10.1016/0013-4694(88)90149-6).
- [14] Fardin Ghorbani, Soheil Hashemi, Ali Abdolali, and Mohammad Soleimani. “EEGsig machine learning-based toolbox for End-to-End EEG signal processing”. In: *arXiv preprint arXiv:2010.12877* (2020).
- [15] Polina Golland and Bruce Fischl. “Permutation tests for classification: towards statistical significance in image-based studies”. In: *Proceedings of IPMI’03: The 18th International Conference on Information Processing in Medical Imaging*. Springer, 2003, pp. 330–341.
- [16] J C Gower and G J S Ross. “Minimum Spanning Trees and Single Linkage Cluster Analysis”. In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 18.1 (1969), pp. 54–64.

- [17] Alexandre Gramfort, Martin Luessi, Eric Larson, Denis Engemann, Daniel Strohmeier, Christian Brodbeck, Roman Goj, Mainak Jas, Teon Brooks, Lauri Parkkonen, and Matti Hämäläinen. “MEG and EEG data analysis with MNE-Python”. In: *Frontiers in Neuroscience* 7 (2013), p. 267. DOI: [10.3389/fnins.2013.00267](https://doi.org/10.3389/fnins.2013.00267).
- [18] Alexandre Gramfort, Martin Luessi, Eric Larson, Denis A Engemann, Daniel Strohmeier, Christian Brodbeck, Lauri Parkkonen, and Matti S Hämäläinen. “MNE software for processing MEG and EEG data”. In: *NeuroImage* 86 (2014), pp. 446–460. DOI: <http://dx.doi.org/10.1016/j.neuroimage.2013.10.027>.
- [19] Iris IA Groen, Sennay Ghebreab, Victor AF Lamme, and H Steven Scholte. “Low-level contrast statistics are diagnostic of invariance of natural textures”. In: *Frontiers in Computational Neuroscience* 6 (2012), p. 34.
- [20] Tijl Grootswagers, Amanda K Robinson, and Thomas A Carlson. “The representational dynamics of visual objects in rapid serial visual processing streams”. In: *NeuroImage* 188 (2019), pp. 668–679.
- [21] Tijl Grootswagers, Susan G Wardle, and Thomas A Carlson. “Decoding dynamic brain patterns from evoked responses: A tutorial on multivariate pattern analysis applied to time series neuroimaging data”. In: *Journal of cognitive neuroscience* 29.4 (2017), pp. 677–697.
- [22] Dominik Grotegerd, Ronny Redlich, Jorge R C Almeida, Mona Riemenschneider, Harald Kugel, Volker Arolt, and Udo Dannlowski. “MANIA—A Pattern Classification Toolbox for Neuroimaging Data”. In: *Neuroinformatics* 12.3 (2014), pp. 471–486. DOI: [10.1007/s12021-014-9223-8](https://doi.org/10.1007/s12021-014-9223-8).
- [23] Matthias Guggenmos, Philipp Sterzer, and Radoslaw Martin Cichy. “Multivariate pattern analysis for MEG: A comparison of dissimilarity measures”. In: *Neuroimage* 173 (2018), pp. 434–447.
- [24] Sebastian Halder, Massimiliano Rea, R Andreoni, Femke Nijboer, Eva Maria Hammer, Sonja Claudia Kleih, Niels Birbaumer, and A Kübler. “An auditory oddball brain–computer interface for binary choices”. In: *Clinical Neurophysiology* 121.4 (2010), pp. 516–523.
- [25] Michael Hanke, Yaroslav O Halchenko, Per B Sederberg, Emanuele Olivetti, Ingo Fründ, Jochem W Rieger, Christoph S Herrmann, James V Haxby, Stephen J Hanson, and Stefan Pollmann. “PyMVPA: a unifying approach to the analysis of neuroscientific data”. In: *Frontiers in neuroinformatics* 3 (2009), p. 337.
- [26] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. 2nd ed. Springer, 2009.

- [27] Stefan Haufe, Frank Meinecke, Kai Grger, Sven Dhne, John-Dylan Haynes, Benjamin Blankertz, and Felix Biemann. “On the interpretation of weight vectors of linear models in multivariate neuroimaging”. In: *Neuroimage* 87 (2014), pp. 96–110.
- [28] Christopher R Holdgraf, Jochem W Rieger, Cristiano Micheli, Stephanie Martin, Robert T Knight, and Frederic E Theunissen. “Encoding and decoding models in cognitive electrophysiology”. In: *Frontiers in systems neuroscience* 11 (2017), p. 61.
- [29] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. “A practical guide to support vector classification”. In: (2003).
- [30] Gopal Chandra Jana, Abhishek Karmakar, and Anupam Agrawal. “EEG VMAC Toolbox: A User-Friendly Open-Source Toolbox for EEG Signals Visualization Manipulation Analysis and Classification”. In: *Procedia Computer Science* 258 (2025), pp. 1062–1080.
- [31] Blair Kaneshiro, Steinunn Arnardttir, Anthony M Norcia, and Patrick Suppes. “Object Category EEG Dataset (OCED)”. In: *Stanford Digital Repository*. 2015.
- [32] Blair Kaneshiro, Marcos Perreau Guimaraes, Hyung-Suk Kim, Anthony M. Norcia, and Patrick Suppes. “A Representational Similarity Analysis of the Dynamics of Object Processing Using Single-Trial EEG Classification”. In: *PLOS ONE* 10.8 (Aug. 2015), pp. 1–27. DOI: [10.1371/journal.pone.0135697](https://doi.org/10.1371/journal.pone.0135697).
- [33] Blair Kaneshiro, Marcos Perreau Guimaraes, Hyung-Suk Kim, Anthony M. Norcia, and Patrick Suppes. “EEG data analyzed in “A Representational Similarity Analysis of the Dynamics of Object Processing Using Single-Trial EEG Classification””. In: *Stanford Digital Repository*. <http://purl.stanford.edu/bq914sc3730>. 2015.
- [34] Roozbeh Kiani, Hossein Esteky, Koorosh Mirpour, and Keiji Tanaka. “Object category structure in response patterns of neuronal population in monkey inferior temporal cortex”. In: *Journal of neurophysiology* 97.6 (2007), pp. 4296–4309.
- [35] Do-Won Kim, Han-Jeong Hwang, Jeong-Hwan Lim, Yong-Ho Lee, Ki-Young Jung, and Chang-Hwan Im. “Classification of selective attention to auditory stimuli: Toward vision-free brain–computer interfacing”. In: *Journal of Neuroscience Methods* 197.1 (2011), pp. 180–185. DOI: <http://doi.org/10.1016/j.jneumeth.2011.02.007>.
- [36] Nathan C L Kong, Blair Kaneshiro, Daniel L K Yamins, and Anthony M Norcia. “Time-resolved correspondences between deep neural network layers and EEG measurements in object processing”. In: *Vision Research* 172 (2020), pp. 27–45.
- [37] Christian Andreas Kothe and Scott Makeig. “BCILAB: A platform for brain–computer interface development”. In: *Journal of Neural Engineering* 10.5 (2013), p. 056014.
- [38] Mario M Krell, Sirko Straube, Anett Seeland, Hendrik Whrle, Johannes Teiwes, Jan H Metzen, Elsa A Kirchner, and Frank Kirchner. “pySPACE—a signal processing and classification environment in Python”. In: *Frontiers in Neuroinformatics* 7 (2013), p. 40.

- [39] Nikolaus Kriegeskorte, Marieke Mur, and Peter A Bandettini. “Representational similarity analysis - connecting the branches of systems neuroscience”. In: *Frontiers in Systems Neuroscience* 2.4 (2008). DOI: [10.3389/neuro.06.004.2008](https://doi.org/10.3389/neuro.06.004.2008).
- [40] Nikolaus Kriegeskorte, Marieke Mur, Douglas A Ruff, Roozbeh Kiani, Jerzy Bodurka, Hossein Esteky, Keiji Tanaka, and Peter A Bandettini. “Matching categorical object representations in inferior temporal cortex of man and monkey”. In: *Neuron* 60.6 (2008), pp. 1126–1141. DOI: <http://dx.doi.org/10.1016/j.neuron.2008.10.043>.
- [41] Karl M Kuntzleman, Jacob M Williams, Phui Cheng Lim, Ashok Samal, Prahalada K Rao, and Matthew R Johnson. “Deep-learning-based multivariate pattern analysis (dMVP): A tutorial and a toolbox”. In: *Frontiers in Human Neuroscience* 15 (2021), p. 638052.
- [42] Yiwen Li, Mingming Zhang, Shuaicheng Liu, and Wenbo Luo. “EEG decoding of multi-dimensional information from emotional faces”. In: *NeuroImage* 258 (2022), p. 119374.
- [43] David López-García, Jose MG Peñalver, Juan M Górriz, and María Ruz. “MVPAlab: A machine learning decoding toolbox for multidimensional electroencephalography data”. In: *Computer Methods and Programs in Biomedicine* 214 (2022), p. 106549.
- [44] Steven Losorelli, Blair Kaneshiro, Gabriella A Musacchia, Nikolas H Blevins, and Matthew B Fitzgerald. “Factors influencing classification of frequency following responses to speech and music stimuli”. In: *Hearing Research* 398 (2020), p. 108101.
- [45] Steven Losorelli, Blair Kaneshiro, Gabriella A Musacchia, Karanvir Singh, Nikolas H Blevins, and Matthew B Fitzgerald. “Stanford Translational Auditory Research Laboratory—Frequency following response dataset 1 (STAR-FFR-01)”. In: *Stanford digital repository*. 2019.
- [46] Ilkka Muukkonen, Kaisu Ölander, Jussi Numminen, and Viljami R Salmela. “Spatio-temporal dynamics of face perception”. In: *NeuroImage* 209 (2020), p. 116531.
- [47] Hamed Nili, Cai Wingfield, Alexander Walther, Li Su, William Marslen-Wilson, and Nikolaus Kriegeskorte. “A Toolbox for Representational Similarity Analysis”. In: *PLOS Computational Biology* 10.4 (Apr. 2014), pp. 1–11. DOI: [10.1371/journal.pcbi.1003553](https://doi.org/10.1371/journal.pcbi.1003553).
- [48] Quentin Noirhomme, Damien Lesenfans, Francisco Gomez, Andrea Soddu, Jessica Schrouff, Gaëtan Garraux, André Luxen, Christophe Phillips, and Steven Laureys. “Biased binomial assessment of cross-validated estimation of classification accuracies illustrated in diagnosis predictions”. In: *NeuroImage: Clinical* 4 (2014), pp. 687–694.
- [49] Robert Oostenveld, Pascal Fries, Eric Maris, and Jan-Mathijs Schoffelen. “FieldTrip: open source software for advanced analysis of MEG, EEG, and invasive electrophysiological data”. In: *Computational intelligence and neuroscience* 2011.1 (2011), p. 156869.

- [50] Nikolaas N Oosterhof, Andrew C Connolly, and James V Haxby. “CoSMoMvPA: multi-modal multivariate pattern analysis of neuroimaging data in Matlab/GNU Octave”. In: *Frontiers in neuroinformatics* 10 (2016), p. 27.
- [51] G Pfurtscheller and C Neuper. “Motor imagery and direct brain-computer communication”. In: *Proceedings of the IEEE* 89.7 (2001), pp. 1123–1134. DOI: [10.1109/5.939829](https://doi.org/10.1109/5.939829).
- [52] Jessica Schrouff, Maria J Rosa, Jane M Rondina, Andre F Marquand, Carlton Chu, John Ashburner, Christophe Phillips, Jonas Richiardi, and Janaina Mourao-Miranda. “PRoNTTo: pattern recognition for neuroimaging toolbox”. In: *Neuroinformatics* 11.3 (2013), pp. 319–337. DOI: [10.1007/s12021-013-9178-1](https://doi.org/10.1007/s12021-013-9178-1).
- [53] Roger N Shepard. “Stimulus and response generalization: Deduction of the generalization gradient from a trace model.” In: *Psychological Review* 65.4 (1958), pp. 242–256.
- [54] Roger N Shepard. “Stimulus and response generalization: Tests of a model relating generalization to distance in psychological space”. In: *Journal of Experimental Psychology* 55.6 (1958), pp. 509–523.
- [55] Roger N Shepard. “The analysis of proximities: Multidimensional scaling with an unknown distance function. II”. English. In: *Psychometrika* 27.3 (1962), pp. 219–246. DOI: [10.1007/BF02289621](https://doi.org/10.1007/BF02289621).
- [56] Irina Simanova, Marcel van Gerven, Robert Oostenveld, and Peter Hagoort. “Identifying object categories from event-related EEG: Toward decoding of conceptual representations”. In: *PLoS ONE* 5.12 (Dec. 2010), e14465. DOI: [10.1371/journal.pone.0014465](https://doi.org/10.1371/journal.pone.0014465).
- [57] Li Su, Elisabeth Fonteneau, William Marslen-Wilson, and Nikolaus Kriegeskorte. “Spatiotemporal searchlight representational similarity analysis in EMEG source space”. In: *2012 International Workshop on Pattern Recognition in NeuroImaging (PRNI)*. 2012, pp. 97–100. DOI: [10.1109/PRNI.2012.26](https://doi.org/10.1109/PRNI.2012.26).
- [58] Li Su, Isma Zulfiqar, Fawad Jamshed, Elisabeth Fonteneau, and William Marslen-Wilson. “Mapping tonotopic organization in human temporal cortex: representational similarity analysis in EMEG source space”. In: *Frontiers in Neuroscience* Volume 8 - 2014 (2014). DOI: [10.3389/fnins.2014.00368](https://doi.org/10.3389/fnins.2014.00368).
- [59] Patrick Suppes, Marcos Perreau Guimaraes, and Dik Kin Wong. “Partial orders of similarity differences invariant between EEG-recorded brain and perceptual representations of language.” In: *Neural Computation* 21.11 (2009), pp. 3228–3269.
- [60] Warren S Torgerson. “Multidimensional scaling: I. Theory and method”. English. In: *Psychometrika* 17.4 (1952), pp. 401–419. DOI: [10.1007/BF02288916](https://doi.org/10.1007/BF02288916).

- [61] Bernard C Wang, Raymond Gifford, Nathan C L Kong, Feng Ruan, Anthony M Norcia, and Blair Kaneshiro. “Example data for MatClassRSA v2 release”. In: *Stanford Digital Repository*. <https://purl.stanford.edu/kv831rr3606>. 2025. DOI: [10 . 25740 / kv831rr3606](https://doi.org/10.25740/kv831rr3606).
- [62] Bernard C Wang, Raymond Gifford, Nathan C L Kong, Feng Ruan, Anthony M Norcia, and Blair Kaneshiro. “MatClassRSA v2 release: A MATLAB toolbox for M/EEG classification, proximity matrix construction, and visualization”. In: *bioRxiv* (2025). DOI: [10.1101/2025.11.19.689115](https://doi.org/10.1101/2025.11.19.689115).
- [63] Bernard C Wang, Anthony M Norcia, and Blair Kaneshiro. “MatClassRSA: A Matlab toolbox for M/EEG classification and visualization of proximity matrices”. In: *bioRxiv* (2017). DOI: [10.1101/194563](https://doi.org/10.1101/194563).
- [64] Changming Wang, Shi Xiong, Xiaoping Hu, Li Yao, and Jiacai Zhang. “Combining features from ERP components in single-trial EEG for discriminating four-category visual objects”. In: *Journal of Neural Engineering* 9.5 (2012), p. 056013.
- [65] Susan G Wardle, Nikolaus Kriegeskorte, Tijl Grootswagers, Seyed-Mahdi Khaligh-Razavi, and Thomas A Carlson. “Perceptual similarity of visual patterns predicts dynamic neural activation patterns measured with MEG”. In: *Neuroimage* 132 (2016), pp. 59–70.
- [66] Vincent Weber, Simon Ruch, Nicole H Skieresz, Nicolas Rothen, and Thomas P Reber. “Correlates of implicit semantic processing as revealed by representational similarity analysis applied to EEG”. In: *iScience* 27.11 (2024).
- [67] Siying Xie, Stefanie Hoehl, Merle Moeskops, Ezgi Kayhan, Christian Kliesch, Bert Turtleton, Moritz Köster, and Radoslaw M Cichy. “Visual category representations in the infant brain”. In: *Current Biology* 32.24 (2022), pp. 5422–5432.