

# **MatClassRSA**

**User Manual**

**Version 1.0**

Bernard C. Wang, Anthony M. Norcia, and Blair Kaneshiro

October 2017

# Table of Contents

|  |           |
|--|-----------|
| <b>Introduction</b>  | <b>03</b> |
| <b>Setup and Installation</b>  | <b>05</b> |
| <b>Overview</b>  | <b>06</b> |
| <b>EEG Classification: <code>classifyEEG</code></b>  | <b>08</b> |
| <b>Computing Representational Dissimilarity Matrices:<br/><code>computeRDM</code></b>  | <b>16</b> |
| <b>Visualization Functions: <code>plotMatrix</code>, <code>plotMDS</code>,<br/><code>plotDendrogram</code>, <code>plotMST</code></b> | <b>19</b> |
| <b>Selected References</b>   | <b>29</b> |

# Introduction

## Conditions of Use

**MatClassRSA** is a free MATLAB software; users are free to redistribute and/or modify it under the terms of [The 3-Clause BSD License \(New BSD License\)](#) published in the public domain.

The terms of the license are as follows:

Copyright 2017 Bernard C. Wang, Anthony M. Norcia, and Blair Kaneshiro

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

When **MatClassRSA** is used for research, please consider citing the publication describing the software package:

Bernard C. Wang, Anthony M. Norcia, and Blair Kaneshiro (2017). MatClassRSA: A Matlab Toolbox for M/EEG Classification and Visualization of Proximity Matrices. [bioRxiv preprint 194563](#); doi: <https://doi.org/10.1101/194563>.

For commercial use of **MatClassRSA** please contact Bernard C. Wang at:  
[bernardcwang@gmail.com](mailto:bernardcwang@gmail.com)

## Support:

For questions, comments, suggestions, feature requests, and bug reports, please contact  
[bernardcwang@gmail.com](mailto:bernardcwang@gmail.com)

The EEG data for the example analyses presented here can be downloaded from the Stanford Digital Repository: <https://purl.stanford.edu/bq914sc3730>

## Author Backgrounds

- Bernard C. Wang is an Affiliated Researcher with the Music Engagement Research Initiative (MERI) in the Center for Computer Research in Music and Acoustics (CCRMA) at Stanford University.
- Anthony M. Norcia is a Professor (Research) in the Department of Psychology at Stanford University.
- Blair Kaneshiro is a Research Scientist in the Department of Otolaryngology (Otology/Neurotology division) at Stanford University School of Medicine.

## Software Background

This software was originally developed for members of the Stanford Vision and Neuro-Development Lab, Stanford Translational Auditory Research Lab, and Music Engagement Research Initiative to perform EEG classification in conjunction with visualizations related to Representational Similarity Analysis. Based on the level of interest in the software, we decided to package the functionalities into a toolbox for greater benefit to the general research community.

## Setup and Installation

The software was tested on the following operating systems:

- Linux Fedora 24 (Twenty Four)
- OS X El Capitan (10.11.6)

Software may encounter bugs if run on other versions of Matlab or operating systems than the ones listed above. Please report any bugs to [bernardcwang@gmail.com](mailto:bernardcwang@gmail.com)

### Dependencies

The software is compatible with Matlab 2016b and above. Some functions may be usable on earlier versions of MATLAB, but it has not been fully tested on previous versions.

The software requires two Matlab toolboxes: the Statistics and Machine Learning Toolbox and the Image Processing Toolbox.

The sole external dependency for MatClassRSA is LIBSVM, which is included in the download of MatClassRSA, citation below:

Chih-Chung Chang and Chih-Jen Lin, LIBSVM : a library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2:27:1--27:27, 2011. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>

### Installation:

Download the latest version of this package at <https://github.com/berneezy3/MatClassRSA/>

Upon startup of Matlab, run the following in the Matlab IDE to add the MatClassRSA functions into your search path:

```
> downloadPath = <Your download directory here, in char array format>;  
> addpath([downloadPath 'src/Classification/libsvm-3.21/matlab']);  
> addpath([downloadPath 'src/Classification/']);  
> addpath([downloadPath 'src/RDM_Computation/']);  
> addpath([downloadPath 'src/Visualization']);
```

MatClassRSA functions will be runnable from this point. To automatically import MatClassRSA upon Matlab startup, please create a 'startup.m' somewhere in your Matlab search path (this can be found using the command "path"), and add the above lines into it.

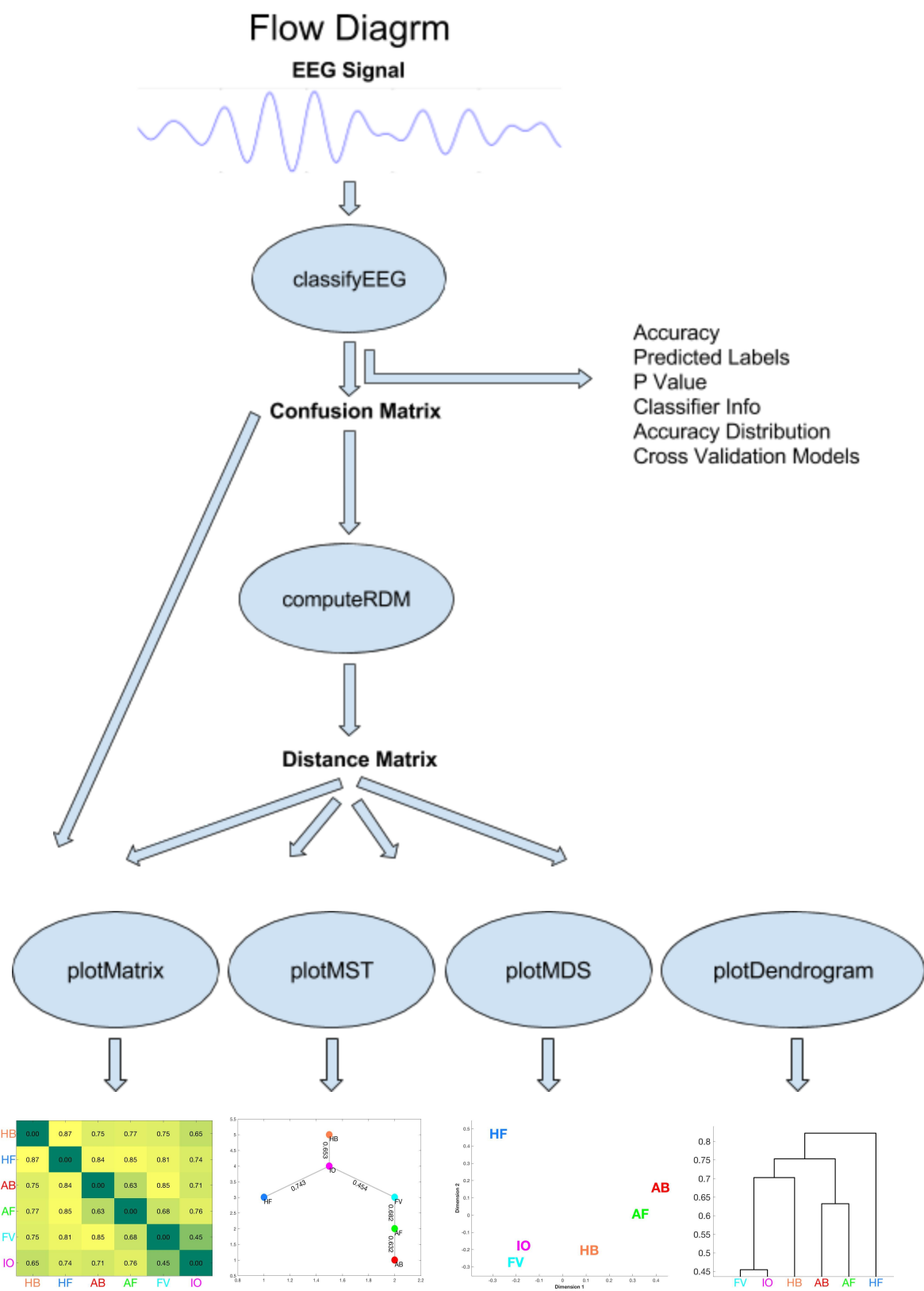
## Overview

M/EEG classification involves the application of machine-learning algorithms to individual or averaged trials of MEG or EEG data and attempting to predict, from the brain response, which stimulus, task attribute, or cognitive state the experimental participant was experiencing. In addition to producing an accuracy (percentage of observations that were correctly assigned a discrete category label), classifiers also produce a confusion matrix which details the actual versus predicted label of each test observations. These confusion matrices can be treated as proximity matrices, which can thus be converted to distance matrices and compared with responses from other response or computational modalities using Representational Similarity Analysis (RSA).

The purpose of this toolbox is to facilitate cognitive neuroscience research using classification. Researchers may wish to classify their M/EEG data to for decoding purposes; to identify features of the brain response that best discriminate stimulus categories; or for classification-based RSA analyses.

The main functionalities of the MatClassRSA toolbox are split into three steps:

1. **Classification:** Classify EEG data to produce a confusion matrix.
2. **Computation of the RDM:** Convert the confusion matrix from step 1 to a Representational Dissimilarity Matrix (RDM).
3. **Visualization:** Visualize RDMs from step 2 as matrix images, dendrograms, and minimum spanning trees, as well as with multidimensional scaling.



## EEG Classification: `classifyEEG`

The first functionality of MatClassRSA is the classification of EEG trials. The classification procedure produces a confusion matrix, which can then be transformed and used for RSA-style visualizations. The `classifyEEG` function handles all classification and cross-validation tasks, and supports numerous related configurations, as described in the below section.

EEG data can be passed in directly to the function, as MatClassRSA supports multiple shapes of data. In addition, the package provides numerous configurable features and options when conducting cross validation,

Correct classifications of each class are plotted on the diagonal, while the off diagonal squares represent misclassifications.

### Usage

```
[CM, accuracy, predY, pVal, classifierInfo, varargout] =  
    classifyEEG(X, Y, Name, Value);
```

### Required Inputs

The classification function has two required inputs: EEG data matrix `X` and labels vector `Y`.

#### `X` — EEG data matrix

The `X` matrix contains the EEG data used for training and testing during classification. The `X` matrix can be passed into the function as the following shapes:

- A 3D space-by-time-by-trials matrix
- A 2D trials-by-features matrix

The input data matrix can be subsetted along the space/time (3D matrix) or features (2D matrix) dimensions using the `'spaceUse'`, `'timeUse'`, and `'featureUse'` name-value pairs described below. 3D input matrices will be reshaped as 2D matrices by the classification function prior to cross validation.

#### `Y` — Labels vector

The `Y` vector contains the numeric labels corresponding to each trial in the 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. Note that classification is the assignment of discrete (category) labels to data observations. The prediction of continuous values, which is a regression task, is not currently implemented in this toolbox.

## Optional Inputs (name-value pair arguments)

### **'spaceUse', 'timeUse', 'featureUse' — Data subsetting (default none)**

For 3D EEG input  $X$  matrices, the user can specify additional name-value parameters 'spaceUse' and 'timeUse'. These should be vectors that contain the indices in the space or feature dimension which the user wants to subset. For example, given a  $128 \times 500 \times 200$  (space \* time \* trials) input matrix  $X$ , the command

```
classifyEEG(X, Y, 'spaceUse', 50:59)
```

would return a  $10 \times 500 \times 200$  matrix, with the  $x$  dimension representing the 50th to 59th space dimension of the original matrix.

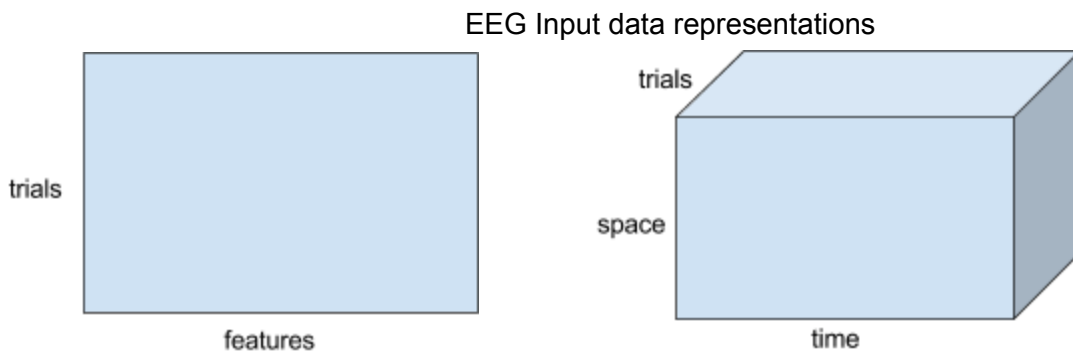
Similarly, with 2D input data matrices, users can pass in the name-value parameter 'featureUse', such that, given a  $4000 \times 5000$  input matrix  $X$ , the command

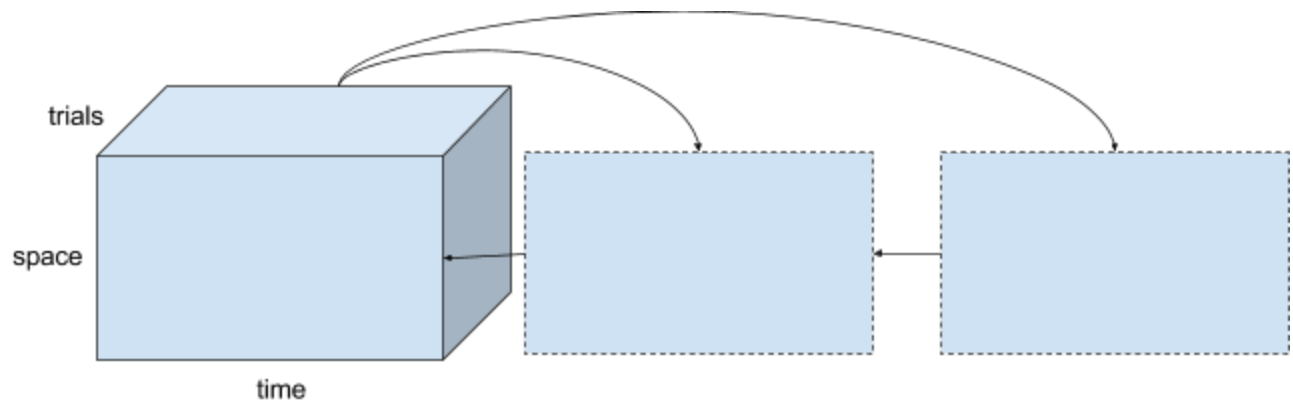
```
classifyEEG(X, Y, 'featureUse', 1:10)
```

would return a  $4000 \times 10$  size matrix, for which the columns now contain the first 10 features of the original input matrix  $X$ .

The default is to not subset the data.

After the optional step of subsetting the matrix, 3D matrices will be converted to 2D matrices by concatenating each trial along the time dimension of the matrix, then passed onto the next step.





### **'randomSeed' — Control randomization (default 'shuffle')**

This parameter controls randomization seeding within the package. The default value, 'shuffle', seeds the random number generator based on current time, which causes execution of the function to return slightly different results each time it is run. If the user wants to obtain replicable results, 'default' should be passed in as the input value, as such

```
classifyEEG(X, Y, 'randomSeed', 'default');
```

### **'shuffleData' — Data shuffling (default 1)**

Once the EEG matrix  $X$  is in the 2D format as input to classification, MatClassRSA by default shuffles the rows in the  $X$  matrix in conjunction with their labels in  $Y$ , such that each row in  $X$  still corresponds to the same label in  $Y$ . The motivation for this operation is to remove possible ordering effects from the classification results when performing cross validation. In cases where the data were collected in blocks, each containing responses to only one stimulus, randomized data shuffling additionally ensures that trials of each class will be evenly distributed throughout folds. However, if input data are already ordered in such a way that the user wants to maintain (e.g., train on data from 9 participants, test on data from the 10th), data shuffling can be turned off.

To turn off data shuffling, add the 'shuffleData' name-value pair as follows:

```
classifyEEG(X, Y, 'shuffleData', 0);
```

### **'averageTrials' — Trial averaging (default none)**

#### **'averageTrialsHandleRemainder' — Handle remainder trials (if any) from trial averaging (default 'discard')**

Trial averaging, or signal averaging, is a technique applied to the EEG trials from the same category to improve the signal to noise ratio. This technique will often improve classification accuracy and produce more distinct structure in visualizations of corresponding confusion

matrices. This is done by grouping trials of the same class together, then replacing each group with their signal average, across the trial dimension.

For example, if we had 50 trials of class 1 and 50 trials of class 2 and decided to average in groups of 5, we would call the function as follows:

```
classifyEEG(X, Y, 'averageTrials', 5);
```

This will split class 1 and class 2 trials each into groups 5 (10 groups), and then average each group across trials, resulting in 10 trials of each class. Cross validation and classification would be performed with these averaged trials.

Sometimes the number of trials of a certain class will not be integer-divisible by the group size that the user specifies. For these cases, MatClassRSA includes an additional parameter, 'averageTrialsHandleRemainder', which accepts these four options to handle remainder cases:

- 'discard': (default) Discard remainder trials.
- 'newGroup': Create new groups using remainder data, even if the groups do not satisfy the user specified number of groups.
- 'append': Append all remainder trials to a group with the same label.
- 'Distribute': Distribute remainder trials among groups with the same label as evenly as possible.

Note: If the 'averageTrialsHandleRemainder' parameter is passed in without a specification for 'averageTrials', the program should output a warning and continue calculations without averaging trials.

### **'PCA' — Principal Component Analysis (default 0.9)**

**'PCAinFold' — Perform Principal Component Analysis separately for each cross-validation training partition (default 1)**

MatClassRSA supports feature dimensionality using Principal Component Analysis (PCA). Here, PCA is performed across the columns (features) of the 2D trials-by-feature EEG matrix  $X$  that is passed into the classifier, transforming the matrix to a trials-by-PC-feature matrix  $X_{PC}$ . Based on the number of PCs retained ( $K$ ), only the first  $K$  columns of  $X_{PC}$  will be input to the classifier.

The number of principal components (PCs) can be selected in two ways. First, the user can specify the exact number of PCs to retain. In this case, an integer value greater than 1 specifies the desired number of PCs. For example, to retain 20 PCs for classification, the function will be called as

```
classifyEEG(X, Y, 'PCA', 20);
```

Alternatively, the user can specify a desired proportion of variance explained, and the corresponding number of PCs (in descending order of variance explained) will be retained for

classification. In this case, a decimal value greater than 0 and less than 1 that represents the proportion of variance the user wants explained should be passed in. For example, to retain as many PCs as are needed to explain 75% of the variance in the data, the user would call

```
classifyEEG(X, Y, 'PCA', 0.75);
```

The default value for 'PCA' is .9, which selects the most important features that explain 90% of the variance. PCA is performed after data subsetting, if subsetting has been specified.

An additional parameter that can be specified is 'PCAinFold'. This parameter allows users to choose whether to conduct PCA on the entire 2D  $X$  matrix (prior to partitioning for cross validation), or separately for each cross-validation training partition. Technically, the correct way to implement PCA would be to compute it separately for every training partition, so that the training observations are not involved in the computation of PCs. However, computing PCA only once on the entire data set is much faster, and in practice often results in similar results as conducting PCA within each separate fold, so we provide that as an option for the user.

'PCAinFold' by default is set to 1 (on), meaning that PCA will be computed separately for every training partition. To compute PCA just once over the entire dataset, call the function as

```
classifyEEG(X, Y, 'PCAinFold', 0);
```

#### **'nFolds' — Number of cross-validation folds (default 10)**

Cross validation is a technique that can be used to assess how a machine learning model trained on a particular dataset will generalize to the population. Cross validation achieves this by splitting the data along  $N$ -folds; then, from  $n = 1:N$ , all the data *except* the observations in fold  $n$  are used to train the model, which is then used to predict the labels of observations in fold  $n$ . The accuracies for each fold are aggregated at the end to produce the estimated accuracy for the classification.

The 'nFolds' parameter must be an integer greater than or equal to 2, and must be less than or equal to the number of trials. The default number of folds is 10. For classifications with very few observations (e.g., less than 100 observations total), it may be wise to increase the number of folds so that more observations can be used for training in each fold. At the most extreme, the user can specify leave-one-out cross validation, where only one observation is withheld for testing in each fold, and therefore the number of folds is equal to the number of observations:

```
classifyEEG(X, Y, 'nFolds', length(Y));
```

**'classify' — Specify classifier type (default 'SVM')**

**'kernel' — Specify decision function kernel for SVM classifier (default 'rbf')**

**'numTrees' - Specify number of decision trees in random Forest classifier (default is 128)**

**'minTreeLeaf' - Specify minimum number of observations per tree leaf (default is 1)**

MatClassRSA currently supports three different classifiers:

- 'SVM': Support Vector Machine (default)
- 'LDA': Linear Discriminant Analysis
- 'RF', 'RandomForest': Random forest

**SVM.** Multi-class classification via Support Vector Machine (SVM) is performed using the LIBSVM library (Chang & Lin, 2011). SVM is the default classifier of MatClassRSA. When SVM is called, the user may additionally specify the input parameter 'kernel', which is a hyperparameter that determines the kernel to use for the decision function in SVM. 'kernel' will not do anything if a classifier other than SVM is used for the classification.

Below are the acceptable kernels:

- 'linear'
- 'polynomial'
- 'rbf' (radial basis function — default)
- 'sigmoid'

For example, to call the SVM classifier with a linear kernel:

```
classifyEEG(X, Y, 'classify', 'SVM', 'kernel', 'linear');
```

**LDA.** Multiclass Linear Discriminant Analysis (LDA) is computed using the `fitcdiscr` function in Matlab. For example,

```
classifyEEG(X, Y, 'classify', 'LDA');
```

**Random Forest.** MatclassRSA uses the `Treebagger` function in Matlab for Random Forest classification. Our `classifyEEG` function allows for the tuning of the following hyperparameters:

- 'numTrees': Number of bagged classification trees to use in the ensemble (default 128).
- 'minLeafSize': Minimum number of observations per tree leaf (default 1).

For example, to use the Random Forest classifier with 50 trees and minimum 2 observations per tree leaf, the user would call the classifier as

```
classifyEEG(X, Y, 'classify', 'RandomForest', ...
    'numTrees', 50, 'minLeafSize', 2);
```

**'pValueMethod' — Specify which method to calculate p-value (default 'binomcdf')**

**'permutations' — Specify how many permutation iterations to perform if doing a permutation test (default 1000).**

We provide 3 ways to compute the p-value of the classification:

- 'binomcdf' (default)
- 'permuteTestLabels'
- 'permuteFullModel'

**Binomial distribution.** The default `'binomcdf'` option uses the classification accuracy and number of test trials per fold to estimate the probability of attaining the mean classifier accuracy under the null hypothesis of the binomial distribution (number of successes  $n$  in  $k$  attempts). Since `'binomcdf'` requires results from only one classification attempt, this is the fastest way to compute p-value. However, this method is not recommended if there exists a high imbalance of observations for each class, or if each test fold contains an insufficient number of observations. Therefore, as a rule of thumb we recommend using this approach with at least 100 observations total, which are split among no more than 10 cross-validation folds (i.e., at least 10 observations in each test partition).

The latter two options conduct permutation testing, in which the number of permutations is determined by the value passed in for the `'permutations'` parameter,  $N$ . This parameter is ignored if the `'binomcdf'` option is used.

**Permute test labels only.** With the `'permuteTestLabels'` option, we perform  $k$ -fold cross validation only once. In each fold, the classification model is trained on the intact data and labels, but predictions are made on test observations whose labels have been shuffled. The prediction is repeated  $N$  times, with the test labels re-randomized for each attempt. The `'permuteTestLabels'` option is the second fastest method, since it requires training the  $k$  models only once, but a total of  $k*N$  prediction operations are performed. So that there are enough test labels to randomize in each fold, here we also recommend having at least 100 observations total, and no more than 10 cross-validation folds.

**Permute full classifier models.** With the `'permuteFullModel'` option, we perform the entire 10-fold cross validation  $N$  times. For each of the  $N$  permutation iterations, the entire labels vector (training and test observations) is shuffled, and in each fold, the classifier model is both trained and tested using the shuffled labels. As the full classification procedure is performed  $N$  times, the `'permuteFullModel'` option is the slowest, but is suitable to use with any classifier configuration, including settings with unbalanced classes, few observations, and up to  $N$ -fold cross validation.

## Outputs

### `'CM'` — Classifier confusion matrix

The confusion matrix summarizes the performance of the classifier. For MatClassRSA confusion matrices, rows represent actual labels while columns represent predicted labels. Element  $i, j$  in the confusion matrix therefore represents the number of observations belonging to class  $i$  that the classifier labeled as belonging to class  $j$ . The sum of the values of the confusion matrix represents the total number of observations classified. Values along the diagonal ( $i=j$ ) represent correct classifications.

### `'accuracy'` — Classifier accuracy

The classifier accuracy is the proportion of classification attempts that were correct. Accuracies are returned as values between 0 and 1 (1 representing 100% accuracy). The accuracy is computed as the total proportion of classifications that were correct (i.e., sum of the diagonal of the confusion matrix divided by the sum of the confusion matrix). Note that for classifications involving unbalanced classes, this manner of computation may not be ideal, and the user may rather prefer to compute a per-class accuracy by normalizing each row of the confusion matrix by the total observations belonging to that class (sum of the row).

### **'predY' — Predicted labels**

This is a vector of predicted labels, in the same order as the input vector  $\mathbf{Y}$ . This vector may be useful for further analyses comparing performance of different classifiers.

### **'pVal' — $p$ -value of the classification**

Scalar value of the  $p$ -value computed according to parameters 'pValueMethod' and 'permutations', as specified above.

### **'classifierInfo' — Configuration option checklist**

This is a struct containing all of the parameters and their respective values that were used for classification.

### **'accDist' - permutation test accuracy distribution**

This is a vector containing all the accuracies computed in the permutation test. Therefore, this output argument only has a valid value if either 'permuteTestLabels' or 'permuteFullModel' is passed in for the parameter 'pValueMethod'. If 'binomcdf' is passed in, then this output will be NaN.

### **'modelsConcat' - Vectorized classification models**

For  $k$ -fold cross validation, the  $k$  number of classification models are saved and output in this struct variable. Note that the models will be different objects depending on the classifier chosen, for example, SVM saves its model into a struct, LDA saves its model into a *ClassificationDiscriminant* object, and RF saves its model into a *TreeBagger* object.

# Computing Representational Dissimilarity Matrices:

## `computeRDM`

Classifier confusions can be thought of as measures of similarity among the brain responses, which can then be related to similarities among the stimuli. That is, similar stimuli may bring about similarities in certain features of the EEG responses; and more similar responses will be more difficult for the classifier to differentiate. Along these lines, the confusion matrix output by a classifier can be treated as a similarity matrix, which can then be converted to a distance matrix, or Representational Dissimilarity Matrix (RDM), and subsequently used in RSA-style analyses.

### Usage

```
RDM = computeRDM(CM, Name, Value);
```

### Required Input

Computation of the RDM requires only one input, the matrix `CM`. We anticipate that for users of this toolbox, the input matrix will typically be the confusion matrix from a classification; however, as documented below, users can input any generalized proximity matrix (including correlation matrices, pairwise classifier accuracies, and pairwise similarity/distance ratings), and use the optional name-value pair arguments to customize the RDM computation procedure.

#### **'CM' — Input matrix**

The `CM` matrix is a square matrix for which element  $i, j$  contains a pairwise similarity or distance measure of  $i$  and  $j$ . As input, the matrix need not be symmetric and need not contain values on the diagonal. If inputting a confusion matrix from outside of MatClassRSA, the user should ensure that the format is consistent with this toolbox, with rows representing actual class, and columns the predicted class.

### Optional Inputs (name-value pair arguments)

The following optional name-value pair arguments reflect the order of operations for computing the RDM: Normalization, then symmetrization, conversion to distance, and finally conversion to rank or percentile rank distance.

#### **'normalize' — Normalization of matrix rows (default 'diagonal')**

This function normalizes a matrix by dividing each row of the matrix by its sum (giving an estimated conditional probability matrix; element  $i, j$  becoming interpretable as  $P(j|i)$ ), by the value of the respective diagonal element in that row (scaling every element in relation to a unity self-similarity measure for every class), or with no normalization. Normalization options are as follows:



- `'diagonal'`: Divide every row of the matrix by the respective diagonal element for that row (e.g., element 2,2 for row 2; element 9,9 for row 9) (default).
- `'sum'`: Divide every row of the matrix by the sum of that row.
- `'none'`: Do not normalize the matrix.

Note: In the present version of the software, specifying `'diagonal'` or `'sum'` when the operation will produce a divisor of zero (e.g., zero on the diagonal or sum of row being zero) will currently produce an error. The user might consider adding a small offset to avoid these errors. A fix will be introduced in a future version of the software.

### **`'symmetrize'` — Symmetrize the matrix (default `'average'`)**

As input matrices need not be symmetric, this operation provides several options for doing so. The result is that element  $i,j$  of the matrix will be equal to element  $j,i$ . The following symmetrization options are supported:

- `'average'`: Compute arithmetic mean,  $x_{out} = (x_{in} + x_{in}^T) / 2$ . This is the default.
- `'geometric'`: Compute geometric mean,  $x_{out} = \sqrt{x_{in} \cdot x_{in}^T}$ .
- `'harmonic'`: Compute harmonic mean,  $x_{out} = 2 \cdot x_{in} \cdot x_{in}^T ./ (x_{in} + x_{in}^T)$ .
- `'none'`: Do not symmetrize the matrix.

### **`'distance'` — Convert similarities to distances (default `'linear'`)**

#### **`'distpower'` — Additional input parameter for `'power'` or `'log'` distance calculation (default 1)**

After normalization and symmetrization, the matrix of similarities can be converted to distances. MatClassRSA currently offers four options for computing distances:

- `'linear'`: Compute linear distance,  $x_{out} = 1 - x_{in}$ . This is the default.
- `'power'`: Compute power distance,  $x_{out} = 1 - x_{in}^{distpower}$
- `'logarithmic'`: Compute log distance,  
 $x_{out} = 1 - \log_2(distPower \cdot x_{in} + 1) ./ \log_2(distPower + 1)$
- `'none'`: Do not convert matrix values from similarities to differences.

**\*\*If using the `'power'` or `'logarithmic'` parameters, the user can additionally specify the power parameter with the name-value pair `'distpower'` (the value is an integer, default 1). For example, to use power distance with a power of 2, call the function as follows:**

```
RDM = computeRDM(CM, 'distance', 'power', 'distpower', 2);
```

### **`'rankdistances'` — Compute rank or percentile rank of distances (default `'none'`)**

Ranked correlations are sometimes preferred for comparing multiple RDMs. Therefore, this function offers three options for converting distances to ranked distances.

- `'none'`: Do not convert distances to rank distances. This is the default.
- `'rank'`: Convert distances to rank distances (uses tied ranks).

- 'percentrank': Convert distances to percentile rank distances.

## Examples

Compute RDM from MatClassRSA confusion matrix, using all default parameters (normalize using diagonal; symmetrize with arithmetic mean; linear distance computation; no ranks):

```
RDM = computeRDM(CM);
```

Compute RDM with no normalization, symmetrize with geometric mean, power 2 distance computation, and rank distance (e.g., input matrix is pairwise similarity ratings):

```
RDM = computeRDM(CM, 'normalize', 'none', 'symmetrize', ...  
                  'geometric', 'distance', 'power', 'distpower', 2, ...  
                  'rankdistances', 'rank');
```

Compute RDM with no normalization or symmetrization; linear distance computation, and percent rank (e.g., input matrix is pairwise correlation matrix):

```
RDM = computeRDM(CM, 'normalize', 'none', ...  
                  'symmetrize', 'none', 'rankdistances', 'percentrank');
```

## Visualization Functions: `plotMatrix`, `plotMDS`, `plotDendrogram`, `plotMST`

Visualizations of the distance matrices computed in the previous section can reveal information about the structure of the stimuli as they are represented in the brain response. MatClassRSA offers four such visualizations: Visualization of a confusion matrix/RDM, dendrogram plot, multidimensional scaling (MDS), and minimum spanning tree (MST). The latter three visualizations in particular may bring to light stimulus clusters that are not immediately evident from visualization of the confusion matrix alone.

### Plot Distance Matrix/Confusion Matrix

To visualize distance matrices or confusion matrices, we use the `plotMatrix()` function.

#### Usage

```
img = plotMatrix(matrix, varargin)
plotMatrix(matrix, varargin)
```

#### Required Input

##### **'RDM' — Input matrix**

RDM - a relational dissimilarity matrix, either computed from the MatClassRSA `computeRDM` function or acquired externally.

#### Optional Inputs (name-value pair arguments)

##### **'axisColors' — Label each class using colored squares (default 'black')**

This parameter can be used in conjunction with `'axisLabels'` (below) to display color-coded labels. Colors should be passed in via a cell array of either color abbreviations, full-length color names, or RGB color triplets.

For example, to set node colors with standard Matlab color abbreviations:

```
plotMatrix(RDM, 'axisColors', {'y' 'm' 'c' 'r'})
```

To set node colors using full-length color names:

```
plotMatrix(RDM, 'axisColors', {'yellow' 'magenta' 'cyan'
    'red'})
```

To set node colors using RGB triplets:

```
plotMatrix(RDM, 'axisColors',...
           {'[1 1 0]','[1 0 1]','[0 1 1]','[1 0 0]'}))
```

**'axisLabels' — Label each class using text (default class numbers)**

This parameter, used in conjunction with 'axisColors' (above), will display color-coded labels. Labels should be passed in via a cell array, where each element in the cell array is a char array representing the string label. For example,

```
plotMatrix(RDM, 'axisLabels', {'a', 'b', 'c', 'd'})
```

**'iconPath' - Label each class using .jpeg or .png files.**

This parameter specifies a directory location in which category labels are stored, *in the same order in which the classes are ordered*. The path should be passed in as the relative or absolute location of the file containing the images. Images will be automatically resized to a square shape. For example,

```
plotMatrix(RDM, 'iconPath', '../Figs/')
```

**'colorMap' - Overall color scheme of the plot**

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')
```

**'colorBar' - Choose whether to display colorbar or not (default 0)**

Set this parameter to 0 to hide, or 1 to show.

**'matrixLabels' - Print matrix values in cells (default 1)**

Use this parameter to choose whether or not to display values for each square in the matrix. Ignore parameter to turn off, enter any value to turn on.

**'FontSize' - Font size of matrix and axis labels**

Enter a numeric value to specify the font size of the matrix labels and the axis labels.



Figure: Example RDM visualization.

## Plot Dendrogram

Dendrograms are commonly used to illustrate the hierarchical clustering of a dataset. MatClassRSA supports agglomerative clustering, the “bottom-up” approach in which each observation (or class, in this case), starts as a cluster, and each pair of clusters are subsequently merged in each level until all the observations form one cluster altogether.

## Usage

```
img = plotDendrogram(matrix, varargin)
plotDendrogram(matrix, varargin)
```

## Required Input

**‘RDM’ — Input matrix**

RDM - a relational dissimilarity matrix, either computed from the MatClassRSA `computeRDM` function or acquired externally.

## Optional Inputs (name-value pair arguments)

**'distMethod' - Cluster distance computation (default 'average')**

Choose algorithm to compute distance between clusters. The toolbox currently offers the following options:

- 'average': Unweighted average distance (UPGMA). This is the default.
- 'centroid': Centroid distance (UPGMC), appropriate for Euclidean distances only
- 'complete': Furthest distance
- 'median': Weighted center of mass distance (WPGMC), appropriate for Euclidean distances only
- 'single': Shortest distance
- 'ward': Inner squared distance (minimum variance algorithm), appropriate for Euclidean distances only
- 'weighted': Weighted average distance (WPGMA)

**'nodeColors' — Label each class using colored squares (default 'black')**

This parameter can be used in conjunction with 'nodeLabels' (below) to display color-coded labels. Colors should be passed in via a cell array of either color abbreviations, full-length color names, or RGB color triplets.

For example, to set node colors with standard Matlab color abbreviations:

```
plotDendrogram(RDM, 'nodeColors', {'y' 'm' 'c' 'r'})
```

To set node colors using full-length color names:

```
plotDendrogram(RDM, 'nodeColors', ...
    {'yellow' 'magenta' 'cyan' 'red'})
```

To set node colors using RGB triplets:

```
plotDendrogram(RDM, 'nodeColors', ...
    {'[1 1 0]' '[1 0 1]' '[0 1 1]' '[1 0 0]'})
```

**'nodeLabels' — Label each class using text (default class number)**

This parameter, used in conjunction with 'nodeColors' (above), will display color-coded labels. Labels should be passed in via a cell array, where each element in the cell array is a char array representing the string label. For example,

```
plotDendrogram(RDM, 'nodeLabels', {'a', 'b', 'c', 'd'})
```

**'iconPath' - Label each class using .jpeg or .png files.**

This parameter specifies a directory location in which category labels are stored, *in the same order in which the classes are ordered*. The path should be passed in as the relative or absolute

location of the file containing the images. Images will be automatically resized to a square shape. For example,

```
plotDendrogram(RDM, 'iconPath', '../Figs/')
```

#### **'orientation' - Dendrogram orientation (default 'down')**

This parameter lets the user specify which direction to point the dendrogram (orientation defined here as the side that contains the dendrogram leaves).

- 'down' (default)
- 'up'
- 'left'
- 'right'

#### **'reorder' - Specify order of classes in the dendrogram**

This parameter allows the user to pass in an array that contains the preferred order of classes in the dendrogram display. This may cause overlapping dendrograph lines.

#### **'yLim' - Set range of the Y-axis.**

Pass in as an array of length 2, [yMin yMax].

#### **'textRotation' - Rotate text (default 0)**

Set this parameter to an amount in degrees to rotate the text.

#### **'lineWidth' - Line width**

Use this parameter to set the width of the lines in the dendrogram.

#### **'lineColor' - Line color**

Use this parameter to set the color of the lines in the dendrogram. Similar to 'nodeColors', we can either pass in color abbreviations, full-length color names, or RGB color triplets.

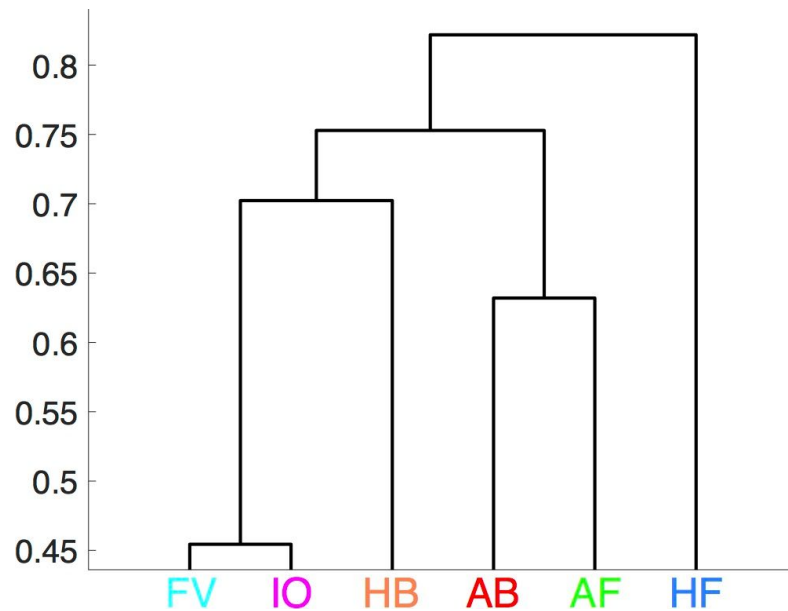


Figure: Example dendrogram.

## Multidimensional Scaling (MDS) Plot

Multidimensional scaling is a visualization technique that projects the distances among items in a set to a lower-dimensional subspace for visualization. In the context of MatClassRSA, MDS will be used to visualize similarity between stimulus categories, as represented by the classifier confusions of the brain response.

### Usage

```
img = plotMDS(matrix, varargin)
plotMDS(matrix, varargin)
```

### Required Input

**'RDM' — Input matrix**

RDM - a relational dissimilarity matrix, either computed from the MatClassRSA `computeRDM` function or acquired externally.

### Optional Inputs (name-value pair arguments)

**'nodeColors' — Label each class using colored squares (default black)**



This parameter can be used in conjunction with 'nodeLabels' (below) to display color-coded labels. Colors should be passed in via a cell array of either color abbreviations, full-length color names, or RGB color triplets.

For example, to set node colors with standard Matlab color abbreviations:

```
plotMDS(RDM, 'nodeColors', {'y' 'm' 'c' 'r'})
```

To set node colors using full-length color names:

```
plotMDS(RDM, 'nodeColors', ...
        {'yellow' 'magenta' 'cyan' 'red'})
```

To set node colors using RGB triplets:

```
plotMDS(RDM, 'nodeColors', ...
        {'[1 1 0]' '[1 0 1]' '[0 1 1]' '[1 0 0]'})
```

### **'nodeLabels' — Label each class using text (default class number)**

This parameter, used in conjunction with 'nodeColors' (above), will display color-coded labels. Labels should be passed in via a cell array, where each element in the cell array is a char array representing the string label. For example,

```
plotMDS(RDM, 'nodeLabels', {'a', 'b', 'c', 'd'})
```

### **'iconPath' - Label each class using .jpeg or .png files.**

This parameter specifies a directory location in which category labels are stored, *in the same order in which the classes are ordered*. The path should be passed in as the relative or absolute location of the file containing the images. Images will be automatically resized to a square shape. For example,

```
plotMDS(RDM, 'iconPath', '../Figs/')
```

### **'dimensions' - Choose which MDS dimensions to display (default [1 2])**

This parameter takes in a vector of length 2, with the first value being the x-axis dimension, the second being the y-axis dimension. There will be  $n$  number of dimensions, with  $n$  being the number of classes (or the number of rows in RDM). Dimension 1 provides the highest-variance separation of data, with  $n$  providing the lowest.

### **'xLim' - Set range of the X-axis**

This parameter takes in an array of length 2, [xMin xMax].

### **'yLim' - Set range of the Y-axis**

This parameter takes in an array of length 2, [yMin yMax].

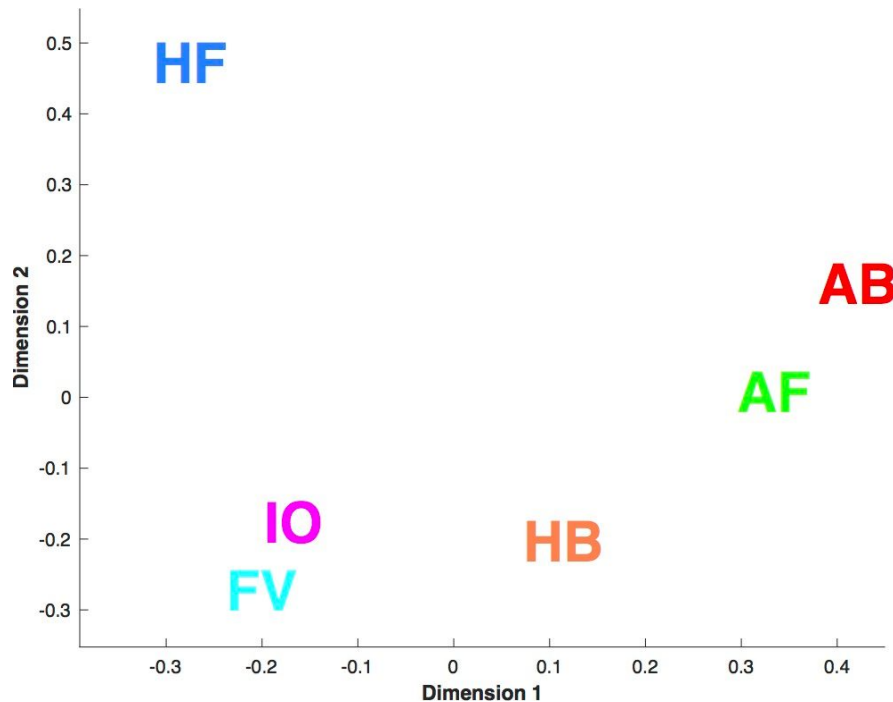


Figure: Example MDS plot.

## Minimum Spanning Tree (MST) Plot

Minimum Spanning Trees are graphs that contain nodes representing each instance of a dataset, with the edges that have the shortest possible cumulative distance with all of the nodes in the graph being connected.

## Usage

```
img = plotMST(matrix, varargin)
plotMST(matrix, varargin)
```

## Required Input

**'RDM'** — Input matrix

RDM - a relational dissimilarity matrix, either computed from the MatClassRSA `computeRDM` function or acquired externally.

## Optional Inputs (name-value pair arguments)

**'nodeColors'** — Label each class using colored squares (default black)

This parameter can be used in conjunction with `'nodeLabels'` (below) to display color-coded labels. Colors should be passed in via a cell array of either color abbreviations, full-length color names, or RGB color triplets.

For example, to set node colors with standard Matlab color abbreviations:

```
plotMST(RDM, 'nodeColors', {'y' 'm' 'c' 'r'})
```

To set node colors using full-length color names:

```
plotMST(RDM, 'nodeColors', ...
        {'yellow' 'magenta' 'cyan' 'red'})
```

To set node colors using RGB triplets:

```
plotMST(RDM, 'nodeColors', ...
        {[1 1 0]' '[1 0 1]' '[0 1 1]' '[1 0 0]'})
```

#### **'nodeLabels' — Label each class using text (default class number)**

This parameter, used in conjunction with `'nodeColors'` (above), will display color-coded labels. Labels should be passed in via a cell array, where each element in the cell array is a char array representing the string label. For example,

```
plotMST(RDM, 'nodeLabels', {'a', 'b', 'c', 'd'})
```

#### **'iconPath' - Label each class using .jpeg or .png files.**

This parameter specifies a directory location in which category labels are stored, *in the same order in which the classes are ordered*. The path should be passed in as the relative or absolute location of the file containing the images. Images will be automatically resized to a square shape. For example,

```
plotMST(RDM, 'iconPath', '../Figs/')
```

#### **'edgeLabelSize' - Size of labels on each edge (default 15)**

Use this parameter to adjust the text size of the labels on each edge of the minimum spanning tree.

#### **'nodeLabelSize' - Size of labels on each node (default 15)**

Use this parameter to adjust the text size of the labels on each node of the minimum spanning tree.

#### **'nodeLabelRotation' - Degrees of rotation for node labels (default 50)**

Use this parameter to specify text rotation, in degrees, of the node labels.

**‘roundEdgeLabel’ - Number of significant figures for edge label display (default 4)**

Use this parameter to specify the number of significant figures by which edge labels should be rounded.

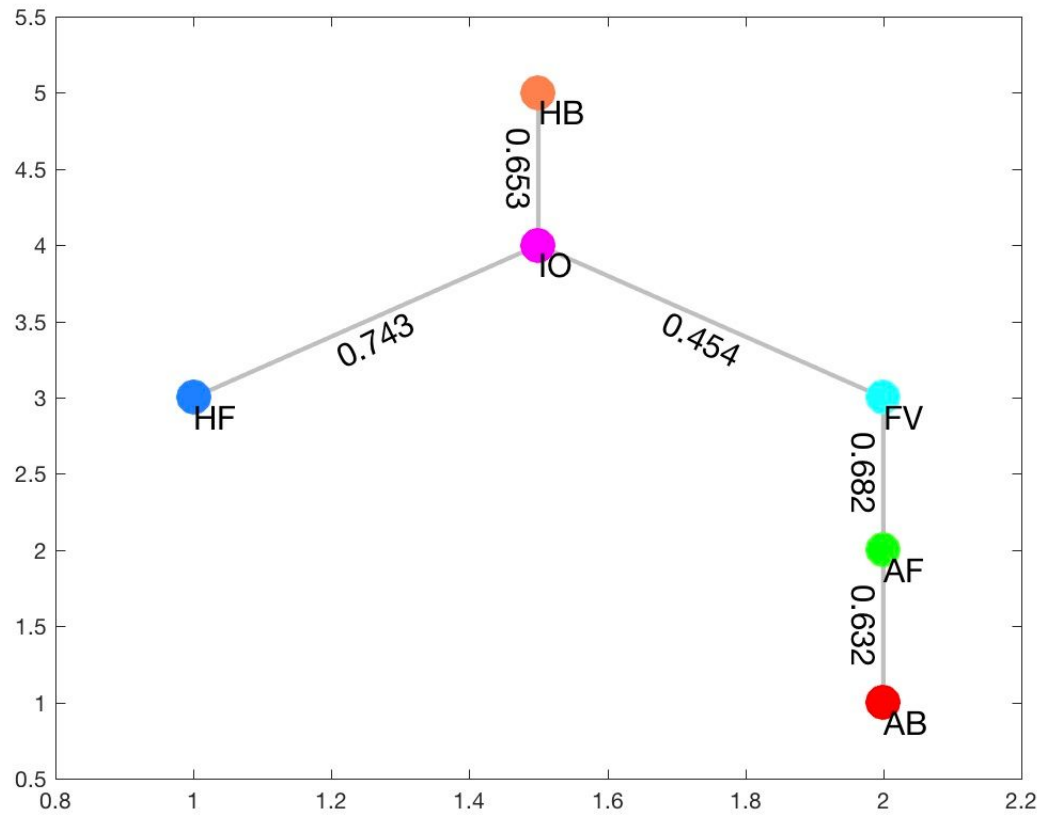


Figure: Example minimum spanning tree.

## Selected References

### M/EEG classification in cognitive neuroscience

Carlson, T., Tovar, D. A., Alink, A., & Kriegeskorte, N. (2013). Representational dynamics of object vision: the first 1000 ms. *Journal of vision*, 13(10), 1-1.

Cichy, R. M., Pantazis, D., & Oliva, A. (2014). Resolving human object recognition in space and time. *Nature neuroscience*, 17(3), 455-462.

Kaneshiro, B., Guimaraes, M. P., Kim, H. S., Norcia, A. M., & Suppes, P. (2015). A representational similarity analysis of the dynamics of object processing using single-trial EEG classification. *PloS one*, 10(8), e0135697.

Perreau Guimaraes, M., Wong, D. K., Uy, E. T., Grosenick, L., & Suppes, P. (2007). Single-trial classification of MEG recordings. *IEEE Transactions on Biomedical Engineering*, 54(3), 436-443.

Philiastides, M. G., & Sajda, P. (2006). Temporal characterization of the neural correlates of perceptual decision making in the human brain. *Cerebral cortex*, 16(4), 509-518.

Schaefer, R. S., Farquhar, J., Blokland, Y., Sadakata, M., & Desain, P. (2011). Name that tune: decoding music from the listening brain. *NeuroImage*, 56(2), 843-849.

Stober, S., Cameron, D. J., & Grahn, J. A. (2014). Classifying EEG Recordings of Rhythm Perception. *Proceedings of the 15th International Society for Music Information Retrieval Conference (ISMIR)*, 649-654.

### Classification

Breiman, L. (2001). Random forests. *Machine learning*, 45(1), 5-32.

Chang, C. C., & Lin, C. J. (2011). LIBSVM: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3), 27.

Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20(3), 273-297.

Fisher, R. A. (1936). The use of multiple measurements in taxonomic problems. *Annals of human genetics*, 7(2), 179-188.

### Representational Similarity Analysis

Kriegeskorte, N., Mur, M., Ruff, D. A., Kiani, R., Bodurka, J., Esteky, H., Tanaka, T., & Bandettini, P. A. (2008). Matching categorical object representations in inferior temporal cortex of man and monkey. *Neuron*, 60(6), 1126-1141.

Kriegeskorte, N., Mur, M., & Bandettini, P. (2008). Representational similarity analysis—connecting the branches of systems neuroscience. *Frontiers in systems neuroscience*, 2.

Nili, H., Wingfield, C., Walther, A., Su, L., Marslen-Wilson, W., & Kriegeskorte, N. (2014). A toolbox for representational similarity analysis. *PLoS computational biology*, 10(4), e1003553.