NeuroImaging with scikit-learn

Release 0.1

Gaël Varoquaux and Alexandre Abraham

http://nisl.github.com

November 21, 2012

Contents

Introduction

1.1 Machine Learning in NeuroImaging: what and why

1.1.1 Machine learning problems and vocabulary

Machine learning is interested in learning from data empirical rules to make **predictions**. Two kind of problems appear:

Supervised learning *Supervised learning* (page 15) is interested in predicting an **output variable**, or **target**, y from **data** X. Typically, we start from labeled data (the **training set**) for which we know the y for each instance of X and train an model; this model is then applied to new unlabeled data (the **test set**) to predict the labels. It maybe be:

- a **regression** problem: predicting a continuous quantity such as age
- a **classification** problem: predicting to which class each observations belongs too: patient or control

In neuroimaging, supervised learning is typically used to relate brain images to behavioral or clinical observations.

Unsupervised learning *Unsupervised learning* (page 28) is concerned with data X without any label. It studies the structure of a dataset, for instance **clustering** or extracting latent factors such as independent components.

In neuroimaging, it is typically used to study resting state, or to find sub-populations in diseases.

1.1.2 Why is machine learning relevant NeuroImaging: a few examples

Diagnosis and prognosis Predicting a clinical score from brain imaging with *supervised learning* (page 15) e.g. [Mourao-Miranda 2012][1]

Generalization scores

- Information mapping: using the prediction accuracy of a classifier to test links between brain images and stimuli. (e.g. *searchlight* (page 23)) [Kriegeskorte 2005][2]

[1] http://www.plosone.org/article/info%3Adoi%2F10.1371%2Fjournal.pone.0029482
[2] http://www.pnas.org/content/103/10/3863.short

- Transfer learning: measuring how much an estimator trained on a task generalizes to another task (e.g. discriminating left from right eye movements also discriminates additions from subtractions [Knops 2009][3])

Statistical estimation From a statistical point of view, machine learning implements statistical estimation of models with a large number of parameters. The tricks pulled in machine learning (e.g. regularization) can enable this estimation with a small number of observations [Varoquaux 2012][4]. This usage of machine learning requires some understanding of the models.

Data mining Data-driven exploration of brain images. *Unsupervised learning* (page 28) extracts structure from the data, such as clusters[5] or multivariate decompositions[6] (*latent factors* such as ICA). This may be useful for implementing some form of *density estimation*: learning a probabilistic model of the data (e.g. in [Thirion 2009][7]).

1.2 Python and the scikit-learn: a primer

What is the scikit-learn?

The scikit-learn[a] is a Python library for machine learning. Its strong points are:

- Easy to use and well documented
- Computationally efficient
- Provide wide variety standard machine learning methods for non-experts

[a] http://scikit-learn.org

1.2.1 Installation of the materials useful for this tutorial

Installing scientific Python

The scientific Python tool stack is rich. Installing the different packages needed one after the other takes a lot of time and is not recommended. We recommend that you install a complete distribution:

Windows EPD[8] or PythonXY[9]: both of these distributions come with the scikit-learn installed (do make sure to install the full, non-free, EPD and not EPD-free to get scikit-learn).

MacOSX EPD[10] is the only full scientific Python distribution for Mac (once again you need to install the full, non-free, EPD and not EPD-free to get scikit-learn).

Linux While EPD[11] is available for Linux, most recent linux distributions come with the package that are needed for this tutorial. Ask your system administrator to install, using the distribution package manager, the following packages:

- scikit-learn (sometimes called *sklearn*)
- matplotlib

[3] http://www.sciencemag.org/content/324/5934/1583.short
[4] http://icml.cc/discuss/2012/688.html
[5] http://scikit-learn.org/stable/modules/clustering.html
[6] http://scikit-learn.org/stable/modules/decomposition.html
[7] http://www.springerlink.com/content/7377x70p5515v778/
[8] http://www.enthought.com/products/epd.php
[9] http://code.google.com/p/pythonxy/
[10] http://www.enthought.com/products/epd.php
[11] http://www.enthought.com/products/epd.php

- ipython

Nibabel

Nibabel[12] is an easy to use reader of NeuroImaging data files. It is not included in scientific Python distributions but is required for all the parts of the tutorial. You can install it with the following command:

```
$ easy_install -U --user nibabel
```

Scikit-learn

If scikit-learn is not installed on your computer, and you have a working install of scientific Python packages (numpy, scipy) and a C compiler, you can add it to your scientific Python install using:

```
$ easy_install -U --user scikit-learn
```

1.2.2 Python for Science quickstart

Don't panic. Python is easy. For a full blown introduction to using Python for science, see the scipy lecture notes[13].

We will be using IPython[14], in pylab mode, that provides an interactive scientific environment. Start it with:

```
$ ipython -pylab
```

It's interactive:

```
Welcome to pylab, a matplotlib-based Python environment
For more information, type 'help(pylab)'.

In [1]: 1 + 2*3
Out[1]: 7
```

Note: Prompt: Below we'll be using >>> to indicate input lines If you wish to copy these input lines directly into your *IPython* console without manually excluding each >>>, you can enable *Doctest Mode* with the command

```
%doctest_mode
```

Scientific computing

In Python, to get scientific features, you need to import the relevant libraries:

Numerical arrays

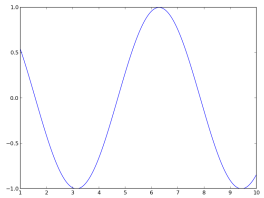
```
>>> import numpy as np
>>> t = np.linspace(1, 10, 2000)  # 2000 points between 1 and 10
>>> t
array([  1.        ,   1.00450225,   1.0090045 , ...,   9.9909955 ,
         9.99549775,  10.        ])
>>> t / 2
```

[12] http://nipy.sourceforge.net/nibabel/
[13] http://scipy-lectures.github.com/
[14] http://ipython.org

```
array([ 0.5       ,  0.50225113,  0.50450225, ...,  4.99549775,
        4.99774887,  5.        ])
>>> np.cos(t) # Operations on arrays are defined in the numpy module
array([ 0.54030231,  0.53650833,  0.53270348, ..., -0.84393609,
       -0.84151234, -0.83907153])
>>> t[:3] # In Python indexing is done with [] and starts at zero
array([ 1.        ,  1.00450225,  1.0090045 ])
```

More documentation...[15]

Plotting



```
>>> import pylab as pl
>>> pl.plot(t, np.cos(t))
[<matplotlib.lines.Line2D object at ...>]
```

More documentation...[16]

Image processing

```
>>> from scipy import ndimage
>>> t_smooth = ndimage.gaussian_filter(t, sigma=2)
```

More documentation...[17]

Signal processing

```
>>> from scipy import signal
>>> t_detrended = signal.detrend(t)
```

More documentation...[18]

Much more

- Simple statistics:

```
>>> from scipy import stats
```

- Linear algebra:

```
>>> from scipy import linalg
```

More documentation...[19]

[15] http://scipy-lectures.github.com/intro/numpy/index.html
[16] http://scipy-lectures.github.com/intro/matplotlib/matplotlib.html
[17] http://scipy-lectures.github.com/advanced/image_processing/index.html
[18] http://scipy-lectures.github.com/intro/scipy.html#signal-processing-scipy-signal
[19] http://scipy-lectures.github.com/intro/scipy.html

Scikit-learn: machine learning

The core concept in the scikit-learn[20] is the estimator object, for instance an SVC (support vector classifier[21]). It is first created with the relevant parameters:

```
>>> from sklearn.svm import SVC
>>> svc = SVC(kernel='linear', C=1.)
```

These parameters are detailed in the documentation of the object: in IPython you can do:

```
In [3]: SVC?
...
Parameters
----------
C : float or None, optional (default=None)
    Penalty parameter C of the error term. If None then C is set
    to n_samples.

kernel : string, optional (default='rbf')
    Specifies the kernel type to be used in the algorithm.
    It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'.
    If none is given, 'rbf' will be used.
...
```

Once the object is created, you can fit it on data, for instance here we use a hand-written digits datasets, that comes with the scikit-learn:

```
>>> from sklearn import datasets
>>> digits = datasets.load_digits()
>>> data = digits.data
>>> labels = digits.target
```

Let's use all but the last 10 samples to train the SVC:

```
>>> svc.fit(data[:-10], labels[:-10])
SVC(C=1.0, ...)
```

and try predicting the labels on the left-out data:

```
>>> svc.predict(data[-10:])
array([ 5.,  4.,  8.,  8.,  4.,  9.,  0.,  8.,  9.,  8.])
>>> labels[-10:]    # The actual labels
array([5, 4, 8, 8, 4, 9, 0, 8, 9, 8])
```

To find out more, try the scikit-learn tutorials[22].

1.2.3 Finding help

Reference material

- A quick and gentle introduction to scientific computing with Python can be found in the scipy lecture notes[23].
- The documentation of the scikit-learn explains each method with tips on practical use and examples: http://scikit-learn.org/ While not specific to neuroimaging, it is often a recommended

[20] http://scikit-learn.org
[21] http://scikit-learn.org/stable/modules/svm.html
[22] http://scikit-learn.org/stable/tutorial/index.html
[23] http://scipy-lectures.github.com/

read. Be careful to consult the documentation relative to the version of the scikit-learn that you are using.

Mailing lists

- You can find help with neuroimaging in Python (file I/O, neuroimaging-specific questions) on the nipy user group: https://groups.google.com/forum/?fromgroups#!forum/nipy-user
- For machine-learning and scikit-learn question, expertise can be found on the scikit-learn mailing list: https://lists.sourceforge.net/lists/listinfo/scikit-learn-general

Getting started: introduction to the decoding pipeline

Nisl comes with code to simplify the use of scikit-learn when dealing with neuroimaging data. For the moment, Nisl is focused on functional MRI data.

Before using a machine learning tool, we may need to apply the following steps:

1. *Data loading and preprocessing* (page **??**) : load Nifti files and check consistency of data
2. *Masking the data* (page **??**) : if a mask is not provided, one is computes automatically
3. *Resampling* (page **??**): optionally we could resampling our data to a different resolution
4. *Temporal Filtering* (page **??**): detrending, regressing out confounds, normalization

2.1 Data loading and preprocessing

2.1.1 Downloading the data

To run demos, the data is retrieved using a function provided by Nisl which will download a dataset and return a bunch of paths to the dataset files (more details in *Downloading example datasets* (page **??**)). We can then proceed loading them as if they were just any other files on our disk. For example, we can download the data from the Haxby 2001 paper:

```
>>> from nisl import datasets
>>> dataset = datasets.fetch_haxby()
```

datasets contains filenames referring to dataset files on the disk:

```
>>> dataset.keys()
['mask_house_little', 'anat', 'mask_house', 'mask_face', 'func', 'session_target', 'mask_vt', 'mask_
>>> dataset.func
['.../haxby2001/subj1/bold.nii.gz']
```

2.1.2 Loading non image data: experiment description

An experiment may need additional information about subjects, sessions or experiments. In the Haxby experiment, fMRI data is acquired while presenting different category of pictures to the subject (face, house...) and the goal of this experiment is to predict which category is presented to the subjects from the brain activation.

These conditions are presented as string into a CSV file. The numpy function *loadtxt* is very useful to load this kind of data.

```
import numpy as np

# Load target information as string and give a numerical identifier to each
labels = np.loadtxt(dataset.session_target[0], dtype=np.str, skiprows=1,
                    usecols=(0,))
index, target = np.unique(labels, return_inverse=True)
```

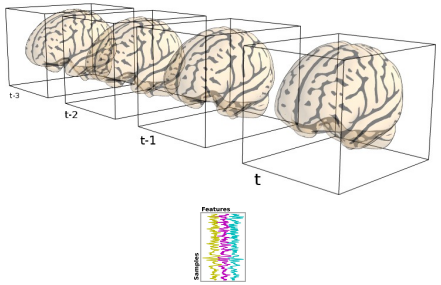
For example, we will now remove the *rest* condition from our dataset. This can be done as follows:

```
no_rest_indices = (labels != 'rest')
target = target[no_rest_indices]
```

Note: If you are not comfortable with this kind of data processing, do not worry: there are plenty of example in Nisl that allows you to easily load data from provided dataset. Do not hesitate to copy/paste the code and adapt it to your own data format if needed. More information can be found in the *data manipulation* (page **??**) section.

2.1.3 Masking the data: from 4D image to 2D array

While the neuroimaging data is made of 4D images, positioned in a coordinate space (which we will call *Niimgs* (page **??**)). For use with the scikit-learn, they need to be converted into, i.e. 2D arrays of samples and features.

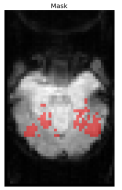


We use masking to convert 4D data (e.g. 3D volume over time) into 2D data (e.g. voxels over time). For this purpose, we use the `NiftiMasker` object, a very powerful data loading tool.

Applying a mask

If your dataset provides a mask, the `NiftiMasker` can apply it automatically. All you have to do is to pass your mask as a parameter when creating your masker. Here we use the mask of the ventral stream, provided with the Haxby dataset.

The `NiftiMasker` can be seen as a *tube* that transforms data from 4D images to 2D arrays, but first it needs to 'fit' this data in order to learn simple parameters from it, such as its shape:




```
from nisl.io import NiftiMasker
nifti_masker = NiftiMasker(mask=dataset.mask_vt[0])

# We give to the nifti_masker a filename, as retrieve a 2D array ready
# for machine learing with scikit-learn
fmri_masked = nifti_masker.fit_transform(dataset.func[0])
```

Note that you can call *nifti_masker.transform(dataset.func[1])* on new data to mask it in a similar way as the data that was used during the fit.

Computing automatically a mask

If your dataset does not provide a mask, the Nifti masker will compute one for you in the *fit* step. The generated mask can be accessed via the *mask_img_* attribute.

Detailed information on automatic mask computation can be found in: *nifti_masker_advanced*.

2.2 Applying a scikit-learn machine learning method

Now that we have a 2D array, we can apply any estimator from the scikit-learn, using its *fit*, *predict* or *transform* methods.

Here, we use scikit-learn Support Vector Classification to learn how to predict the category of picture seen by the subject:

```
# First, we remove rest condition
fmri_masked = fmri_masked[no_rest_indices]

# Here we use a Support Vector Classification, with a linear kernel and C=1
from sklearn.svm import SVC
svc = SVC(kernel='linear', C=1.)

# And we run it
svc.fit(fmri_masked, target)
y_pred = svc.predict(fmri_masked)
```

We will not detail it here since there is a very good documentation about it in the scikit-learn documentation[1]

[1] http://scikit-learn.org/stable/modules/svm.html#classification

2.3 Unmasking (inverse_transform)

Unmasking data is as easy as masking it! This can be done by using method *inverse_transform* on your processed data. As you may want to unmask several kind of data (not only the data that you previously masked but also the results of an algorithm), the masker is clever and can take data of dimension 1D (resp. 2D) to convert it back to 3D (resp. 4D).

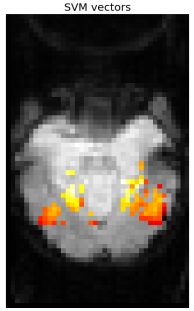
```
# Look at the discriminating weights
sv = svc.support_vectors_

# Reverse masking thanks to the Nifti Masker
niimg = nifti_masker.inverse_transform(sv[0])
```

Here we want to see the disciminating weights of some voxels.

2.4 Visualizing results

Again the visualization code is simple. We can an fMRI slice as a background and plot the weight. Brighter points have a higher discriminating weight.

```
import pylab as pl
import nibabel

# We use a masked array so that the voxels at '-1' are displayed transparently
act = np.ma.masked_array(niimg.get_data(), niimg.get_data() == 0)

### Create the figure
pl.figure()
pl.axis('off')
pl.title('SVM vectors')
pl.imshow(np.rot90(nibabel.load(dataset.func[0]).get_data()[..., 27, 0]),
          interpolation='nearest', cmap=pl.cm.gray)
pl.imshow(np.rot90(act[..., 27]), cmap=pl.cm.hot,
          interpolation='nearest')
```



2.5 Going further

The `NiftiMasker` is a very powerful object and we have only scratched the surface of its possibilities. It is described more in details in the section *The NiftiMasker: loading, masking and filtering* (page **??**). Also, simple functions that can be used to perform elementary operations such as masking or filtering are described in *Preprocessing functions* (page **??**).

CHAPTER 3

Supervised learning

Supervised learning[1] is focussed on predicting on output value. In NeuroImaging it is often used in the context of *decoding*: predicting behavior from brain images. It may also be useful for diagnostic.

3.1 Decoding on simulated data

Objectives

1. Understand linear estimators, (SVM, elastic net, ridge)
2. Use the scikit-learn's linear models

3.1.1 Simple NeuroImaging-like simulations

We simulate data as in Michel et al 2012, *Total variation regularization for fMRI-based prediction of behaviour*, Trans Med Imag: a linear model with a random design matrix $\mathbf{X}$:

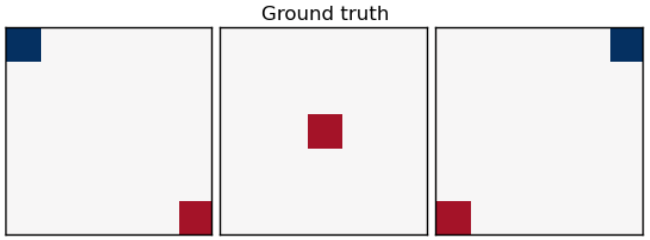
$$\mathbf{y} = \mathbf{X}\mathbf{w} + \mathbf{e}$$

- $\mathbf{w}$: the weights of the linear model correspond to the predictive brain regions. Here, in the simulations, they form a 3D image with 4 regions in opposite corners.
- $\mathbf{X}$: the design matrix corresponds to the observed fMRI data. Here we simulate random normal variables and smooth them as in Gaussian fields.
- $\mathbf{e}$ is random normal noise.

We provide a black-box function to create the data in the *example script*:

```
X_train, X_test, y_train, y_test, snr, noise, coefs, size = \
    create_simulation_data(snr=10, n_samples=400, size=12)
mask = np.ones((size, size, size), np.bool)
process_mask = np.zeros((size, size, size), np.bool)
process_mask[:, :, 0] = True
```

[1] http://en.wikipedia.org/wiki/Supervised_learning




```
process_mask[:, :, 6] = True
process_mask[:, :, 11] = True

coefs = np.reshape(coefs, [size, size, size])
plot_slices(coefs, title="Ground truth")

##############################################################################
```

3.1.2 Running various estimators

We can now run different estimators and look at their prediction score, as well as the feature maps that they recover. Namely, we will use

- A support vector regression (SVM[2])
- An elastic-net[3]
- A *Bayesian* ridge estimator, i.e. a ridge estimator that sets its parameter according to a metaprior
- A ridge estimator that set its parameter by cross-validation

We can create a list with all the estimators readily created with the parameters of our choice:

```
classifiers = [
    ('bayesian_ridge', linear_model.BayesianRidge(normalize=True)),
    ('enet_cv', linear_model.ElasticNetCV(alphas=[5, 1, 0.5, 0.1], rho=0.05)),
    ('ridge_cv', linear_model.RidgeCV(alphas=[100, 10, 1, 0.1], cv=5)),
    ('svr', svm.SVR(kernel='linear', C=0.001)),
    ('searchlight', searchlight.SearchLight(
        mask=mask, process_mask=process_mask,
        masked_data=True,
        radius=4.,
        score_func=r2_score,
        cv=KFold(y_train.size, k=4)))
]
```

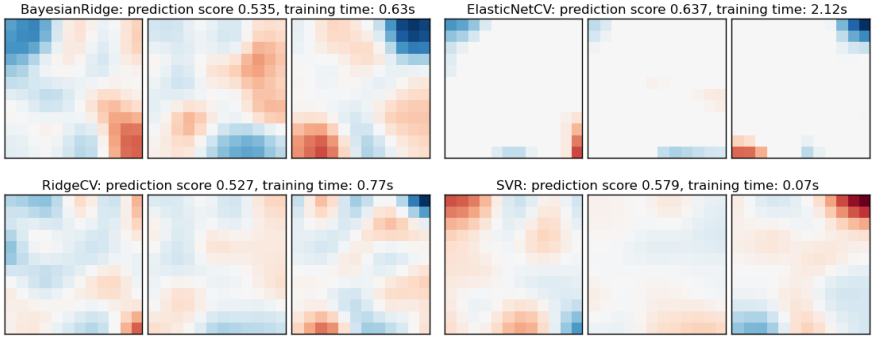
Note that the *RidgeCV* and the *ElasticNetCV* have names ending in *CV* that stands for *cross-validation*: in the list of possible *alpha* values that they are given, they choose the best by cross-validation.

As the estimators expose a fairly consistent API, we can all fit them in a for loop: they all have a *fit* method for fitting the data, a *score* method to retrieve the prediction score, and because they are all linear models, a *coef_* attribute that stores the coefficients $\mathbf{w}$ estimated.

[2] http://scikit-learn.org/stable/modules/svm.html
[3] http://scikit-learn.org/stable/modules/linear_model.html

Note: All parameters estimated from the data end with an underscore

```
for name, classifier in classifiers:
    t1 = time()
    classifier.fit(X_train, y_train)
    elapsed_time = time() - t1

    if name != 'searchlight':
        coefs = classifier.coef_
        coefs = np.reshape(coefs, [size, size, size])
        score = classifier.score(X_test, y_test)
        title = '%s: prediction score %.3f, training time: %.2fs' % (
                classifier.__class__.__name__, score,
                elapsed_time)

    else:  # Searchlight
        coefs = classifier.scores_
        title = '%s: training time: %.2fs' % (
                classifier.__class__.__name__,
                elapsed_time)

    # We use the plot_slices function provided in the example to
    # plot the results
    plot_slices(coefs, title=title)
```



Exercise

Use recursive feature elimination (RFE) with the SVM:

```
>>> from sklearn.feature_selection import RFE
```

Read the object's documentation to find out how to use RFE.
Performance tip: increase the *step* parameter, or it will be very slow.

3.2 fMRI decoding: predicting which objects a subject is viewing

Objectives

At the end of this tutorial you will be able to:
1. Load fMRI volumes in Python.
2. Perform a state-of-the-art decoding analysis of fMRI data.
3. Perform even more sophisticated analyzes of fMRI data.

3.2.1 Data loading and preprocessing

We launch ipython:

```
$ ipython -pylab
```

First, we load the data using the tutorial's data downloader, `nisl.datasets.fetch_haxby_simple` (page **??**):

```
from nisl import datasets
import numpy as np
import nibabel
dataset_files = datasets.fetch_haxby_simple()

# fmri_data and mask are copied to lose the reference to the original data
bold_img = nibabel.load(dataset_files.func)
fmri_data = np.copy(bold_img.get_data())
affine = bold_img.get_affine()
y, session = np.loadtxt(dataset_files.session_target).astype("int").T
conditions = np.recfromtxt(dataset_files.conditions_target)['f0']
mask = dataset_files.mask
# fmri_data.shape is (40, 64, 64, 1452)
# and mask.shape is (40, 64, 64)
```

Then we preprocess the data to make:

- compute the mean of the image to replace anatomic data
- mask the data X and transpose the matrix, so that its shape becomes (n_samples, n_features) (see *From 4D to 2D arrays* (page **??**) for a discussion on using masks)

```
# Build the mean image because we have no anatomic data
mean_img = fmri_data.mean(axis=-1)
```

Exercise

1. Extract the period of activity from the data (i.e. remove the remainder).

Solution

As 'y == 0' in rest, we want to keep only time points for which *y != 0*:

```
>>> X, y, session = X[y!=0], y[y!=0], session[y!=0]
```

Here, we limit our analysis to the *face* and *house* conditions:

```
# Keep only data corresponding to face or houses
condition_mask = np.logical_or(conditions == 'face', conditions == 'house')
```

```
X = fmri_data[..., condition_mask]
y = y[condition_mask]
session = session[condition_mask]
conditions = conditions[condition_mask]

# We have 2 conditions
n_conditions = np.size(np.unique(y))

### Loading step ###############################################################
from nisl.io import NiftiMasker
from nibabel import Nifti1Image
nifti_masker = NiftiMasker(mask=mask, sessions=session, smooth=4)
niimg = Nifti1Image(X, affine)
X = nifti_masker.fit_transform(niimg)
```

3.2.2 Down to business: decoding analysis

Prediction function: the estimator

To perform decoding we construct an estimtor, predicting a condition label **y** given a set **X** of images.

We define here a simple Support Vector Classification[4] (or SVC) with C=1, and a linear kernel. We first import the correct module from scikit-learn and we define the classifier:

```
### Define the prediction function to be used.
# Here we use a Support Vector Classification, with a linear kernel and C=1
from sklearn.svm import SVC
clf = SVC(kernel='linear', C=1.)
```

Need some doc ?

```
>>> clf ?
Type:             SVC
Base Class:       <class 'sklearn.svm.libsvm.SVC'>
String Form:
SVC(kernel=linear, C=1.0, probability=False, degree=3, coef0=0.0, eps=0.001,
cache_size=100.0, shrinking=True, gamma=0.0)
Namespace:        Interactive
Docstring:
    C-Support Vector Classification.
    Parameters
    ----------
    C : float, optional (default=1.0)
        penalty parameter C of the error term.
...
```

Or go to the scikit-learn documentation[5] We use a SVC here, but we can use many other classifiers[6]

Dimension reduction

As there are a very large number of voxels and not all are useful for face vs house prediction, we add a feature selection[7] procedure. The idea is to select the *k* voxels most correlated to the task.

[4] http://scikit-learn.org/stable/modules/svm.html
[5] http://scikit-learn.org/modules/svm.html
[6] http://scikit-learn.org/stable/supervised_learning.html
[7] http://scikit-learn.org/stable/modules/feature_selection.html

For this, we need to import the correct module and define a simple F-score based feature selection (a.k.a. Anova[8]):

```
from sklearn.feature_selection import SelectKBest, f_classif

### Define the dimension reduction to be used.
# Here we use a classical univariate feature selection based on F-test,
# namely Anova. We set the number of features to be selected to 1000
feature_selection = SelectKBest(f_classif, k=500)

# We have our classifier (SVC), our feature selection (SelectKBest), and now,
# we can plug them together in a *pipeline* that performs the two operations
# successively:
from sklearn.pipeline import Pipeline
anova_svc = Pipeline([('anova', feature_selection), ('svc', clf)])
```

Launching it on real data: fit (train) and predict (test)

In scikit-learn, the prediction function has a very simple API:

- a *fit* function that "learns" the parameters of the model from the data. Thus, we need to give some training data to *fit*.
- a *predict* function that "predicts" a target from new data. Here, we just have to give the new set of images (as the target should be unknown):

```
anova_svc.fit(X, y)
y_pred = anova_svc.predict(X)
```

Warning ! Do not do this at home: the prediction that we obtain here is to good to be true (see next paragraph). Here we are just doing a sanity check.

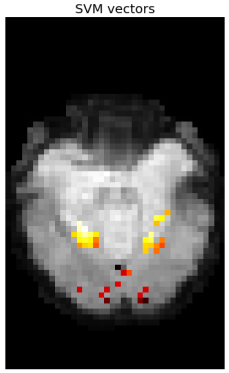
Visualising the results

We can visualize the result of our algorithm:

- we first get the support vectors of the SVC and revert the feature selection mechanism
- we remove the mask
- then we overlay our previously-computed, mean image with our support vectors

```
### Look at the discriminating weights
svc = clf.support_vectors_
# reverse feature selection
svc = feature_selection.inverse_transform(svc)
# reverse masking
niimg = nifti_masker.inverse_transform(svc[0])

# We use a masked array so that the voxels at '-1' are displayed
# transparently
act = np.ma.masked_array(niimg.get_data(), niimg.get_data() == 0)

### Create the figure
import pylab as pl
pl.axis('off')
pl.title('SVM vectors')
pl.imshow(np.rot90(mean_img[..., 27]), cmap=pl.cm.gray,
```

[8] http://en.wikipedia.org/wiki/Analysis_of_variance#The_F-test




```
          interpolation='nearest')
pl.imshow(np.rot90(act[..., 27]), cmap=pl.cm.hot,
          interpolation='nearest')
pl.show()

# Saving the results as a Nifti file may also be important
import nibabel
img = nibabel.NiftilImage(act, affine)
nibabel.save(img, 'haxby_face_vs_house.nii')
```

Cross-validation: measuring prediction performance

However, the last analysis is *wrong*, as we have learned and tested on the same set of data. We need to use a cross-validation to split the data into different sets.

In scikit-learn, a cross-validation is simply a function that generates the index of the folds within a loop. So, now, we can apply the previously defined *pipeline* with the cross-validation:

```
from sklearn.cross_validation import LeaveOneLabelOut

### Define the cross-validation scheme used for validation.
# Here we use a LeaveOneLabelOut cross-validation on the session label
# divided by 2, which corresponds to a leave-two-session-out
cv = LeaveOneLabelOut(session // 2)

### Compute the prediction accuracy for the different folds (i.e. session)
cv_scores = []
for train, test in cv:
    y_pred = anova_svc.fit(X[train], y[train]) \
        .predict(X[test])
    cv_scores.append(np.sum(y_pred == y[test]) / float(np.size(y[test])))
```

But we are lazy people, so there is a specific function, *cross_val_score* that computes for you the results for the different folds of cross-validation:


```
>>> from sklearn.cross_validation import cross_val_score
>>> cv_scores = cross_val_score(anova_svc, X, y, cv=cv)
```

If you are the happy owner of a multiple processors computer you can speed up the computation by using n_jobs=-1, which will spread the computation equally across all processors (this will probably not work under Windows):

```
>>> cv_scores = cross_val_score(anova_svc, X, y, cv=cv, n_jobs=-1, verbose=10)
```

Prediction accuracy We can take a look to the results of the *cross_val_score* function:

```
>>> cv_scores
array([ 1.        ,  1.        ,  1.        ,  1.        ,  1.        ,
        1.        ,  1.        ,  0.94444444,  1.        ,  1.        ,
        1.        ,  1.        ])
```

This is simply the prediction score for each fold, i.e. the fraction of correct predictions on the left-out data.

Exercise

1. Compute the mean prediction accuracy using *cv_scores*

Solution

```
>>> classification_accuracy = np.mean(cv_scores)
>>> classification_accuracy
0.99537037037037035
```

We have a total prediction accuracy of 74% across the different folds.

We can add a line to print the results:

```
### Return the corresponding mean prediction accuracy
classification_accuracy = np.mean(cv_scores)

### Printing the results
print "=== ANOVA ==="
print "Classification accuracy: %f" % classification_accuracy, \
    " / Chance level: %f" % (1. / n_conditions)
# Classification accuracy: 0.986111  / Chance level: 0.500000
```

Final script

The complete script can be found as *an example*. Now, all you have to do is to publish the results :)

3.2.3 Going further with scikit-learn

We have seen a very simple analysis with scikit-learn, but it may be interesting to explore the wide variety of supervised learning algorithms in the scikit-learn[9].

[9] http://scikit-learn.org/stable/supervised_learning.html

Changing the prediction function

We now see how one can easily change the prediction function, if needed. We can try the Linear Discriminant Analysis (LDA)[10]

Import the module:

```
>>> from sklearn.lda import LDA
```

Construct the new prediction function and use it in a pipeline:

```
>>> from sklearn.pipeline import Pipeline
>>> lda = LDA()
>>> anova_lda = Pipeline([('anova', feature_selection), ('LDA', lda)])
```

and recompute the cross-validation score:

```
>>> cv_scores = cross_val_score(anova_lda, X, y, cv=cv, verbose=1)
>>> classification_accuracy = np.mean(cv_scores)
>>> print "Classification accuracy: %f" % classification_accuracy, \
...     " / Chance level: %f" % (1. / n_conditions)
Classification accuracy: 1.000000   / Chance level: 0.500000
```

Changing the feature selection

Let's say that you want a more sophisticated feature selection, for example a Recursive Feature Elimination (RFE)[11]

Import the module:

```
>>> from sklearn.feature_selection import RFE
```

Construct your new fancy selection:

```
>>> rfe = RFE(SVC(kernel='linear', C=1.), 50, step=0.25)
```

and create a new pipeline:

```
>>> rfe_svc = Pipeline([('rfe', rfe), ('svc', clf)])
```

and recompute the cross-validation score:

```
>>> cv_scores = cross_val_score(rfe_svc, X, y, cv=cv, n_jobs=-1,
...     verbose=True)
```

But, be aware that this can take A WHILE...

3.3 Searchlight : finding voxels containing information

3.3.1 Searchlight principle

Searchlight was introduced in Information-based functional brain mapping[12], Nikolaus Kriegeskorte, Rainer Goebel and Peter Bandettini (PNAS 2006) and consists in scanning the images volume with a *searchlight*. Briefly, a ball of given radius is scanned across the brain volume and the prediction accuracy of a classifier trained on the corresponding voxels is measured.

[10] http://scikit-learn.org/auto_examples/plot_lda_qda.html
[11] http://scikit-learn.org/stable/modules/feature_selection.html#recursive-feature-elimination
[12] http://www.pnas.org/content/103/10/3863

3.3.2 Preprocessing

Loading

As seen in *previous sections* (page **??**), fetching the data from internet and loading it can be done with the provided functions:

```
from nisl import datasets
import numpy as np
import nibabel

dataset_files = datasets.fetch_haxby_simple()

# fmri_data and mask are copied to lose the reference to the original data
bold_img = nibabel.load(dataset_files.func)
fmri_data = np.copy(bold_img.get_data())
affine = bold_img.get_affine()
y, session = np.loadtxt(dataset_files.session_target).astype("int").T
conditions = np.recfromtxt(dataset_files.conditions_target)['f0']
mask = dataset_files.mask
```

Preparing data

For this tutorial we need:

- to put X in the form *n_samples* x *n_features*
- compute a mean image for visualisation background

```
# Build the mean image because we have no anatomic data
mean_img = fmri_data.mean(axis=-1)

### Restrict to faces and houses ##############################################
condition_mask = np.logical_or(conditions == 'face', conditions == 'house')
X = fmri_data[..., condition_mask]
y = y[condition_mask]
session = session[condition_mask]
conditions = conditions[condition_mask]

### Loading step ##############################################################
from nisl.io import NiftiMasker
from nibabel import Nifti1Image

nifti_masker = NiftiMasker(mask=mask, sessions=session)
niimg = Nifti1Image(X, affine)
X_masked = nifti_masker.fit(niimg).transform(niimg)
X_preprocessed = nifti_masker.inverse_transform(X_masked).get_data()
X_preprocessed = np.rollaxis(X_preprocessed, axis=-1)
mask = nifti_masker.mask_img_.get_data().astype(np.bool)
```

Masking

One of the main element that distinguich Searchlight from other algorithms is this notion of structuring element that scan the entire volume. If this seems rather intuitive, it has in fact an impact on the masking procedure.

Most of the time, fMRI data is masked and then given to the algorithm. This is not possible in the case of Searchlight because, to compute the score of non-masked voxels, some masked voxels may be needed. This is why two masks will be used here :

- *mask* is the anatomical mask
- *process_mask* is a subset of mask and contains voxels to be processed.

process_mask will then be used to restrain computation to one slice, in the back of the brain. *mask* will ensure that no value outside of the brain is taken into account when iterating with the sphere.

```
process_mask = mask.copy()
process_mask[..., 38:] = False
process_mask[..., :36] = False
process_mask[:, 30:] = False

### Searchlight ###############################################################

# Make processing parallel
# /!\ As each thread will print its progress, n_jobs > 1 could mess up the
#     information output.
n_jobs = 1

### Define the score function used to evaluate classifiers
# Here we use precision which maesures proportion of true positives among
# all positives results for one class.
from sklearn.metrics import precision_score
score_func = precision_score

### Define the cross-validation scheme used for validation.
# Here we use a KFold cross-validation on the session, which corresponds to
# splitting the samples in 4 folds and make 4 runs using each fold as a test
# set once and the others as learning sets
from sklearn.cross_validation import KFold
cv = KFold(y.size, k=4)

### Fit ########################################################################

from nisl import searchlight

# The radius is the one of the Searchlight sphere that will scan the volume
searchlight = searchlight.SearchLight(mask, process_mask, radius=1.5,
        n_jobs=n_jobs, score_func=score_func, verbose=1, cv=cv)

searchlight.fit(X_preprocessed, y)

### Visualization #############################################################
import pylab as pl
pl.figure(1)
# searchlight.scores_ contains per voxel cross validation scores
s_scores = np.ma.array(searchlight.scores_, mask=np.logical_not(process_mask))
pl.imshow(np.rot90(mean_img[..., 37]), interpolation='nearest',
        cmap=pl.cm.gray)
pl.imshow(np.rot90(s_scores[..., 37]), interpolation='nearest',
        cmap=pl.cm.hot, vmax=1)
pl.axis('off')
pl.title('Searchlight')
pl.show()

### Show the F_score
```



```
from sklearn.feature_selection import f_classif
pl.figure(2)
X_masked = X_preprocessed[:, process_mask]
f_values, p_values = f_classif(X_masked, y)
p_values = -np.log10(p_values)
p_values[np.isnan(p_values)] = 0
p_values[p_values > 10] = 10
p_unmasked = np.zeros(mask.shape)
p_unmasked[process_mask] = p_values
p_ma = np.ma.array(p_unmasked, mask=np.logical_not(process_mask))
pl.imshow(np.rot90(mean_img[..., 37]), interpolation='nearest',
        cmap=pl.cm.gray)
pl.imshow(np.rot90(p_ma[..., 37]), interpolation='nearest',
        cmap=pl.cm.hot)
pl.title('F-scores')
pl.axis('off')
pl.show()
```

Restricting the dataset

Like in the *decoding* (page 17) example, we limit our analysis to the *face* and *house* conditions:

```
condition_mask = np.logical_or(conditions == 'face', conditions == 'house')
X = fmri_data[..., condition_mask]
y = y[condition_mask]
session = session[condition_mask]
conditions = conditions[condition_mask]

### Loading step ##############################################################
from nisl.io import NiftiMasker
from nibabel import Nifti1Image

nifti_masker = NiftiMasker(mask=mask, sessions=session)
niimg = Nifti1Image(X, affine)
X_masked = nifti_masker.fit(niimg).transform(niimg)
X_preprocessed = nifti_masker.inverse_transform(X_masked).get_data()
X_preprocessed = np.rollaxis(X_preprocessed, axis=-1)
mask = nifti_masker.mask_img_.get_data().astype(np.bool)

### Prepare the masks #########################################################
# Here we will use several masks :
# * mask is the originalmask
# * process_mask is a subset of mask, it contains voxels that should be
#   processed (we only keep the slice z = 26 and the back of the brain to speed
#   up computation)
process_mask = mask.copy()
process_mask[..., 38:] = False
process_mask[..., :36] = False
process_mask[:, 30:] = False
```

3.3.3 Third Step: Setting up the searchlight

Classifier

The classifier used by default by Searchlight is LinearSVC with C=1 but this can be customed easily by passing an estimator parameter to the cross validation. See scikit-learn documentation for other classifiers[13].

Score function

Here we use precision as metrics to measures proportion of true positives among all positives results for one class. Many others are available in scikit-learn documentation[14].

```
from sklearn.metrics import precision_score
score_func = precision_score
```

Cross validation

Searchlight will iterate on the volume and give a score to each voxel. This score is computed by running a classifier on selected voxels. In order to make this score as accurate as possible (and avoid overfitting), a cross validation is made.

As Searchlight is a little costly, we have chosen a cross validation method that do not take too much time. K-Fold along with $K = 4$ is a good compromise between running time and result.

```
from sklearn.cross_validation import KFold
cv = KFold(y.size, k=4)
```

3.3.4 Running Searchlight

Running Searchlight is straightforward now that everything is set. The only parameter left is the radius of the ball that will run through the data. Kriegskorte uses a 4mm radius because it yielded the best detection performance in his simulation.

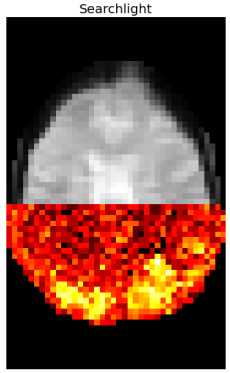
```
from nisl import searchlight

# The radius is the one of the Searchlight sphere that will scan the volume
searchlight = searchlight.SearchLight(mask, process_mask, radius=1.5,
        n_jobs=n_jobs, score_func=score_func, verbose=1, cv=cv)

searchlight.fit(X_preprocessed, y)
```

3.3.5 Visualisation

Searchlight

As the activation map is cropped, we use the mean image of all scans as a background. We can see here that voxels in the visual cortex contains information to distinguish pictures showed to the volunteer, which was the expected result.

[13] http://scikit-learn.org/supervised_learning.html
[14] http://scikit-learn.org/supervised_learning.html



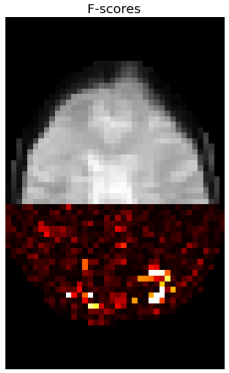

```
import pylab as pl
pl.figure(1)
# searchlight.scores_ contains per voxel cross validation scores
s_scores = np.ma.array(searchlight.scores_, mask=np.logical_not(process_mask))
pl.imshow(np.rot90(mean_img[..., 37]), interpolation='nearest',
        cmap=pl.cm.gray)
pl.imshow(np.rot90(s_scores[..., 37]), interpolation='nearest',
        cmap=pl.cm.hot, vmax=1)
pl.axis('off')
pl.title('Searchlight')
pl.show()
```

Comparing to standard-analysis: F_score or SPM

The standard approach to brain mapping is performed using *statistical parametric mapping* (SPM), using ANOVA (analysis of variance), and F tests. Here we use is to compute the *p-values* of the voxels [15]. To display the results, we use the negative log of the p-value.

```
from sklearn.feature_selection import f_classif
pl.figure(2)
X_masked = X_preprocessed[:, process_mask]
f_values, p_values = f_classif(X_masked, y)
p_values = -np.log10(p_values)
p_values[np.isnan(p_values)] = 0
p_values[p_values > 10] = 10
p_unmasked = np.zeros(mask.shape)
p_unmasked[process_mask] = p_values
p_ma = np.ma.array(p_unmasked, mask=np.logical_not(process_mask))
pl.imshow(np.rot90(mean_img[..., 37]), interpolation='nearest',
        cmap=pl.cm.gray)
pl.imshow(np.rot90(p_ma[..., 37]), interpolation='nearest',
        cmap=pl.cm.hot)
```

[15] The *p-value* is the probability of getting the observed values assuming that nothing happens (i.e. under the null hypothesis). Therefore, a small *p-value* indicates that there is a small chance of getting this data if no real difference existed, so the observed voxel must be significant.




```
pl.title('F-scores')
pl.axis('off')
pl.show()
```


Unsupervised learning

Unsupervised learning[1] is focussed on finding structure in a given data. In NeuroImaging two common tasks are clustering and finding meaningful components (e.g. using ICA).

4.1 fMRI clustering

4.1.1 Resting-state dataset

Here, we use a resting-state[2] dataset from test-retest study performed at NYU. Details on the data can be found in the documentation for the downloading function `fetch_nyu_rest` (page **??**).

4.1.2 Preprocessing: loading and masking

As seen in *previous sections* (page **??**), we fetch the data from internet and load it with a provided function:

```
import numpy as np
from nisl import datasets, io
dataset = datasets.fetch_nyu_rest(n_subjects=1)
nifti_masker = io.NiftiMasker()
fmri_masked = nifti_masker.fit_transform(dataset.func[0])
mask = nifti_masker.mask_img_.get_data().astype(np.bool)
```

No mask is given with the data so we let the masker compute one. The result is a niimg from which we extract a numpy array that is used to mask our original X.

[1] http://en.wikipedia.org/wiki/Unsupervised_learning
[2] http://www.nitrc.org/projects/nyu_trt/

4.1.3 Applying Ward clustering

Compute connectivity map

Before computing the ward itself, we compute a connectivity map. This is useful to constrain clusters to form contiguous parcels (see the scikit-learn documentation[3])

```
from sklearn.feature_extraction import image
shape = mask.shape
connectivity = image.grid_to_graph(n_x=shape[0], n_y=shape[1],
                                   n_z=shape[2], mask=mask)
```

Principle

The Ward algorithm is a hierarchical clustering algorithm: it successfully merges together voxels that have similar timecourses.

Caching

Note that in practice the scikit-learn implementation of the Ward clustering first computes a tree of possible merges, and then, the requested number of clusters breaks it apart the tree at the right level.

As no matter how many clusters we want, we do not need to compute the tree again, we can rely on caching to speed things up when varying the number of cluster. Scikit-learn integrates a transparent caching library (joblib[4]). In the ward clustering, the *memory* parameter is used to cache the computed component tree. You can give it either a *joblib.Memory* instance or the name of directory used for caching.

Apply the ward

Here we simply launch the ward to find 1000 clusters and we time it.

```
from sklearn.cluster import WardAgglomeration
import time
start = time.time()
ward = WardAgglomeration(n_clusters=1000, connectivity=connectivity,
                         memory='nisl_cache')
ward.fit(fmri_masked)
print "Ward agglomeration 1000 clusters: %.2fs" % (time.time() - start)
```

This runs in about 10 seconds (depending on your computer configuration). Now, we are not satisfied of the result and we want to cluster the picture in 2000 elements.

```
# the caching mechanism
start = time.time()
ward = WardAgglomeration(n_clusters=2000, connectivity=connectivity,
                         memory='nisl_cache')
ward.fit(fmri_masked)
print "Ward agglomeration 2000 clusters: %.2fs" % (time.time() - start)
```

Now that the component tree has been computed, computation is must faster thanks to caching. You should have the result in less than 1 second.

[3] http://www.scikit-learn.org/stable//modules/clustering.html#adding-connectivity-constraints
[4] http://packages.python.org/joblib/

4.1.4 Post-Processing and visualization

Unmasking

After applying the ward, we must unmask the data. This can be done simply :

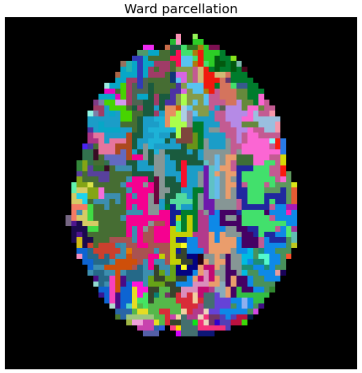
```
# Avoid 0 label
labels = ward.labels_ + 1
labels = nifti_masker.inverse_transform(ward.labels_).get_data()
# 0 is the background, putting it to -1
labels = labels - 1
```

You can see that masked data is filled with -1 values. This is done for the sake of visualization. In fact, clusters are labeled with going from 0 to (n_clusters - 1). By putting every other values to -1, we assure that uninteresting values will not mess with the visualization.

Label visualization

We can visualize the clusters. We assign random colors to each cluster for the labels visualization.

```
import pylab as pl

# Cut at z=20
cut = labels[:, :, 20].astype(int)
# Assign random colors to each cluster. For this we build a random
# RGB look up table associating a color to each cluster, and apply it
# below
import numpy as np
colors = np.random.random(size=(ward.n_clusters + 1, 3))
# Cluster '-1' should be black (it's outside the brain)
colors[-1] = 0
pl.figure()
pl.axis('off')
pl.imshow(colors[np.rot90(cut)], interpolation='nearest')
pl.title('Ward parcellation')
```

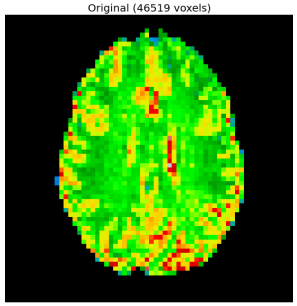


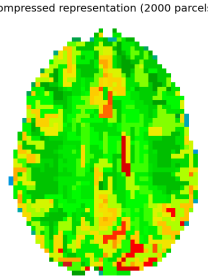
Compressed picture

By transforming a picture in a new one in which the value of each voxel is the mean value of the cluster it belongs to, we are creating a compressed version of the original picture. We can obtain this representation thanks to a two step procedure :

- call *ward.transform* to obtain the mean value of each cluster (for each scan)
- call *ward.inverse_transform* on the previous result to turn it back into the masked picture shape

```
pl.figure()
first_epi = nifti_masker.inverse_transform(fmri_masked[0]).get_data()
first_epi = np.ma.masked_array(first_epi, first_epi == 0)
# Outside the mask: a uniform value, smaller than inside the mask
first_epi[np.logical_not(mask)] = 0.9 * first_epi[mask].min()
vmax = first_epi[..., 20].max()
vmin = first_epi[..., 20].min()
pl.imshow(np.rot90(first_epi[..., 20]),
          interpolation='nearest', cmap=pl.cm.spectral, vmin=vmin, vmax=vmax)
pl.axis('off')
pl.title('Original (%i voxels)' % fmri_masked.shape[1])

# A reduced data can be create by taking the parcel-level average:
# Note that, as many objects in the scikit-learn, the ward object exposes
# a transform method that modifies input features. Here it reduces their
# dimension
fmri_reduced = ward.transform(fmri_masked)

# Display the corresponding data compressed using the parcellation
fmri_compressed = ward.inverse_transform(fmri_reduced)
compressed = nifti_masker.inverse_transform(
    fmri_compressed[0]).get_data()
compressed = np.ma.masked_equal(compressed, 0)


pl.figure()
pl.imshow(np.rot90(compressed[:, :, 20]),
          interpolation='nearest', cmap=pl.cm.spectral, vmin=vmin, vmax=vmax)
pl.title('Compressed representation (2000 parcels)')
pl.axis('off')
pl.show()
```





We can see that using only 2000 parcels, we can approximate well the original image.

4.2 ICA of resting-state fMRI datasets

Independent Analysis of resting-state fMRI data is useful to extract brain networks in an unsupervised maner (data-driven):

- Kiviniemi et al, *Independent component analysis of nondeterministic fMRI signal sources*, Neuroimage 2009
- Beckmann et al, *Investigations into resting-state connectivity using independent component analysis*, Philos Trans R Soc Lond B 2005

4.2.1 Preprocessing

Loading

As seen in *previous sections* (page **??**), we fetch the data from internet and load it with a provided function:

```
from nisl import datasets
# Here we use only 3 subjects to get faster-running code. For better
# results, simply increase this number
dataset = datasets.fetch_nyu_rest(n_subjects=1)
# XXX: must get the code to run for more than 1 subject
```

Conctenating, smoothing and masking

```
from nisl import io

masker = io.NiftiMasker(smooth=8)
data_masked = masker.fit_transform(dataset.func[0])

# Concatenate all the subjects
#fmri_data = np.concatenate(data_masked, axis=1)
fmri_data = data_masked

# Take the mean along axis 3: the direction of time
mean_img = masker.inverse_transform(fmri_data.mean(axis=0))
```

4.2.2 Applying ICA

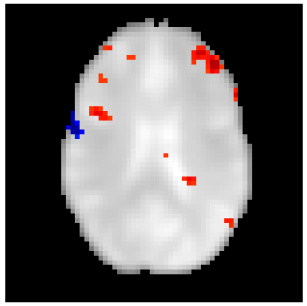
```
from sklearn.decomposition import FastICA
n_components = 20
ica = FastICA(n_components=n_components, random_state=42)
components_masked = ica.fit_transform(data_masked.T).T

# We normalize the estimated components, for thresholding to make sens
components_masked -= components_masked.mean(axis=0)
components_masked /= components_masked.std(axis=0)
# Threshold
components_masked[np.abs(components_masked) < 1.3] = 0

# Now we inverting the masking operation, to go back to a full 3D
# representation
component_img = masker.inverse_transform(components_masked)
components = component_img.get_data()
```

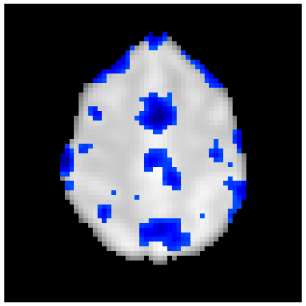
```
# Using a masked array is important to have transparency in the figures
components = np.ma.masked_equal(components, 0, copy=False)
```

4.2.3 Visualizing the results

Visualization follows similarly as in the previous examples. Remember that we use masked arrays (*np.ma*) to create transparency in the overlays.

```
# Show some interesting components
mean_epi = mean_img.get_data()
import pylab as pl
pl.figure()
pl.axis('off')
vmax = np.max(np.abs(components[:, :, 19, 15]))
pl.imshow(np.rot90(mean_epi[:, :, 19]), interpolation='nearest',
          cmap=pl.cm.gray)
pl.imshow(np.rot90(components[:, :, 19, 15]), interpolation='nearest',
          cmap=pl.cm.jet, vmax=vmax, vmin=-vmax)

pl.figure()
pl.axis('off')
vmax = np.max(np.abs(components[:, :, 25, 19]))
pl.imshow(np.rot90(mean_epi[:, :, 25]), interpolation='nearest',
          cmap=pl.cm.gray)
pl.imshow(np.rot90(components[:, :, 25, 19]), interpolation='nearest',
          cmap=pl.cm.jet, vmax=vmax, vmin=-vmax)
pl.show()
```





Note: Note that as the ICA components are not ordered, the two components displayed on your computer might not match those of the tutorial. For a fair representation, you should display all the components and investigate which one resemble those displayed above.

fMRI data manipulation: input/output, masking, visualization...

5.1 Downloading example datasets

This tutorial package embeds tools to download and load datasets. They can be imported from `nisl.datasets` (page **??**):

```
>>> from nisl import datasets
>>> haxby_files = datasets.fetch_haxby_simple()
>>> # The structures contains paths to haxby dataset files:
>>> haxby_files.keys()
['data', 'session_target', 'mask', 'conditions_target']
>>> import nibabel
>>> haxby_data = nibabel.load(haxby_files.func)
>>> haxby_data.get_data().shape # 1452 time points and a spatial size of 40x64x64
(40, 64, 64, 1452)
```

`fetch_haxby` (page **??**)([data_dir, n_subjects, url, ...])	Download and loads complete haxby dataset
`fetch_haxby_simple` (page **??**)([data_dir, url, resume, ...])	Download and loads an example haxby dataset
`fetch_nyu_rest` (page **??**)([n_subjects, sessions, ...])	Download and loads the NYU resting-state test-retest dataset
`fetch_adhd` (page **??**)([n_subjects, data_dir, url, ...])	Download and loads the ADHD resting-state dataset

5.1.1 nisl.datasets.fetch_haxby

`nisl.datasets.`**`fetch_haxby`**(*data_dir=None*, *n_subjects=1*, *url=None*, *resume=True*, *verbose=0*)

Download and loads complete haxby dataset

Parameters **data_dir: string, optional** :

Path of the data directory. Used to force data storage in a specified location. Default: None

n_subjects: integer, optional :

Number of subjects, from 1 to 5.

Returns **data: Bunch** :

Dictionary-like object, the interest attributes are : 'anat': string list

Paths to anatomic images

'func': string list Paths to nifti file with bold data

'session_target': string list Paths to text file containing session and target data

'mask_vt': string list Paths to nifti ventral temporal mask file

'mask_face': string list Paths to nifti ventral temporal mask file

'mask_house': string list Paths to nifti ventral temporal mask file

'mask_face_little': string list Paths to nifti ventral temporal mask file

'mask_house_little': string list Paths to nifti ventral temporal mask file

Notes

PyMVPA provides a tutorial using this dataset : http://www.pymvpa.org/tutorial.html

More informations about its structure : http://dev.pymvpa.org/datadb/haxby2001.html

See additional information[1]

References

Haxby, J., Gobbini, M., Furey, M., Ishai, A., Schouten, J., and Pietrini, P. (2001). Distributed and overlapping representations of faces and objects in ventral temporal cortex. Science 293, 2425-2430.

5.1.2 nisl.datasets.fetch_haxby_simple

`nisl.datasets.`**`fetch_haxby_simple`**(*data_dir=None*, *url=None*, *resume=True*, *verbose=0*)

Download and loads an example haxby dataset

Parameters **data_dir: string, optional** :

Path of the data directory. Used to force data storage in a specified location. Default: None

Returns **data: Bunch** :

Dictionary-like object, the interest attributes are : 'func': string

Path to nifti file with bold data

'session_target': string Path to text file containing session and target data

'mask': string Path to nifti mask file

'session': string Path to text file containing labels (can be used for LeaveOneLabelOut cross validation for example)

[1] http://www.sciencemag.org/content/293/5539/2425

Notes

PyMVPA provides a tutorial using this dataset : http://www.pymvpa.org/tutorial.html

More informations about its structure : http://dev.pymvpa.org/datadb/haxby2001.html

See additional information[2]

References

Haxby, J., Gobbini, M., Furey, M., Ishai, A., Schouten, J., and Pietrini, P. (2001). Distributed and overlapping representations of faces and objects in ventral temporal cortex. Science 293, 2425-2430.

5.1.3 nisl.datasets.fetch_nyu_rest

`nisl.datasets.`**`fetch_nyu_rest`**(*n_subjects=None, sessions=[1], data_dir=None, verbose=0*)

Download and loads the NYU resting-state test-retest dataset

Parameters **n_subjects: integer optional** :

The number of subjects to load. If None is given, all the subjects are used.

n_sessions: array of integers optional :

The sessions to load. Load only the first session by default.

data_dir: string, optional :

Path of the data directory. Used to force data storage in a specified location. Default: None

Returns **data** : Bunch

Dictionary-like object, the interest attributes are : 'func': string list

Paths to functional images

'anat_anon': string list Paths to anatomic images

'anat_skull': string Paths to skull-stripped images

'session': numpy array List of ids corresponding to images sessions

Notes

This dataset is composed of 3 sessions of 26 participants (11 males). For each session, three sets of data are available:

- anatomical:
  - anonymized data (defaced thanks to BIRN defacer)
  - skullstripped data (using 3DSkullStrip from AFNI)
- functional

For each participant, 3 resting-state scans of 197 continuous EPI functional volumes were collected :

- 39 slices

[2] http://www.sciencemag.org/content/293/5539/2425

•matrix = 64 x 64

•acquisition voxel size = 3 x 3 x 3 mm

Sessions 2 and 3 were conducted in a single scan session, 45 min apart, and were 5-16 months after Scan 1.

All details about this dataset can be found here : http://cercor.oxfordjournals.org/content/19/10/2209.full

References

Documentation http://www.nitrc.org/docman/?group_id=274

Download http://www.nitrc.org/frs/?group_id=274

Paper to cite The Resting Brain: Unconstrained yet Reliable[3] Z. Shehzad, A.M.C. Kelly, P.T. Reiss, D.G. Gee, K. Gotimer, L.Q. Uddin, S.H. Lee, D.S. Margulies, A.K. Roy, B.B. Biswal, E. Petkova, F.X. Castellanos and M.P. Milham.

Other references

- The oscillating brain: Complex and Reliable[4] X-N. Zuo, A. Di Martino, C. Kelly, Z. Shehzad, D.G. Gee, D.F. Klein, F.X. Castellanos, B.B. Biswal, M.P. Milham
- Reliable intrinsic connectivity networks: Test-retest evaluation using ICA and dual regression approach[5], X-N. Zuo, C. Kelly, J.S. Adelstein, D.F. Klein, F.X. Castellanos, M.P. Milham

5.1.4 nisl.datasets.fetch_adhd

`nisl.datasets.`**`fetch_adhd`**(*n_subjects=None*, *data_dir=None*, *url=None*, *resume=True*, *verbose=0*)

Download and loads the ADHD resting-state dataset

Parameters **n_subjects: integer optional** :

The number of subjects to load. If None is given, all the 40 subjects are used.

data_dir: string, optional :

Path of the data directory. Used to force data storage in a specified location. Default: None

url: string, optional :

Override download URL. Used for test only (or if you setup a mirror of the data).

Returns **data** : Bunch

Dictionary-like object, the interest attributes are : 'func': string list

Paths to functional images

'parameters': string list Parameters of preprocessing steps

[3] http://cercor.oxfordjournals.org/content/19/10/2209
[4] http://dx.doi.org/10.1016/j.neuroimage.2009.09.037
[5] http://dx.doi.org/10.1016/j.neuroimage.2009.10.080

References

Download

ftp://www.nitrc.org/fcon_1000/htdocs/indi/adhd200/sites/ ADHD200_40sub_preprocessed.tgz

The data are downloaded only once and stored locally, in order:

- the folder specified by *data_dir* parameter in the fetching function if it is specified
- the environment variable *NISL_DATA* if it exists
- the *nisl_data* folder in the current directory

Note that you can copy that folder across computers to avoid downloading.

5.2 Understanding MRI data: Nifti or analyze files

NIfTI and Analyse file structures

NifTi[a] files (or Analyze files) are the standard way of sharing data in neuroimaging. We may be interested in the following three main components:

data raw scans bundled in a numpy array: *data = img.get_data()*
affine gives the correspondance between voxel index and spatial location: *affine = img.get_affine()*
header informations about the data (slice duration...): *header = img.get_header()*

[a] http://nifti.nimh.nih.gov/

Neuroimaging data can be loaded simply thanks to nibabel[6]. Once the file is downloaded, a single line is needed to load it.

```
from nisl import datasets
haxby_files = datasets.fetch_haxby_simple()

# Get the file names relative to this dataset
bold = haxby_files.func

# Load the NIfTI data
import nibabel
nifti_img = nibabel.load(bold)
fmri_data = nifti_img.get_data()
```

Dataset formatting: data shape

We can find two main representations for MRI scans:

- a big 4D matrix representing 3D MRI along time, stored in a big 4D NifTi file. FSL[a] users tend to prefer this format.
- several 3D matrices representing each volume (time point) of the session, stored in set of 3D Nifti or analyse files. SPM[b] users tend to prefer this format.

[a] http://www.fmrib.ox.ac.uk/fsl/
[b] http://www.fil.ion.ucl.ac.uk/spm/

[6] http://nipy.sourceforge.net/nibabel/

5.3 Visualizing brain images

Once that NIfTI data is loaded, visualization is simply the display of the desired slice (the first three dimensions) at a desired time point (fourth dimension). For *haxby*, data is rotated so we have to turn each image counter clockwise.

```
import numpy as np
import pylab as pl

# Compute the mean EPI: we do the mean along the axis 3, which is time
mean_img = np.mean(fmri_data, axis=3)

# pl.figure() creates a new figure
pl.figure()

# First subplot: coronal view
# subplot: 1 line, 3 columns and use the first subplot
pl.subplot(1, 3, 1)
# Turn off the axes, we don't need it
pl.axis('off')
# We use pl.imshow to display an image, and use a 'gray' colormap
# we also use np.rot90 to rotate the image
pl.imshow(np.rot90(mean_img[:, 32, :]), interpolation='nearest',
          cmap=pl.cm.gray)
pl.title('Coronal')

# Second subplot: sagittal view
pl.subplot(1, 3, 2)
pl.axis('off')
pl.title('Sagittal')
pl.imshow(np.rot90(mean_img[15, :, :]), interpolation='nearest',
          cmap=pl.cm.gray)

# Third subplot: axial view
pl.subplot(1, 3, 3)
pl.axis('off')
pl.title('Axial')
pl.imshow(np.rot90(mean_img[:, :, 32]), interpolation='nearest',
          cmap=pl.cm.gray)
```

5.4 The `NiftiMasker`: loading, masking and filtering

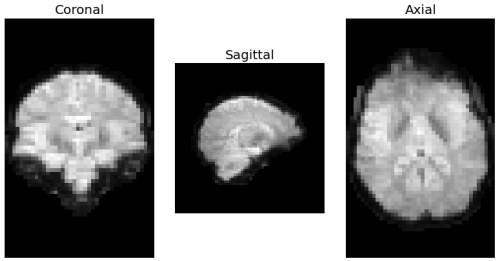
In this section gives some details and show how to custom data loading. For this, we rely on the `NiftiMasker` class. Advanced usage of this class uses its parameters to tweak algorithms. Sometimes, you will have to go beyond it and put your hands in the code to achieve what you want.

The `NiftiMasker` is a power tool to *1)* load data easily, *2)* preprocess it and then *3)* send it directly into a scikit-learn pipeline. It is designed to apply some basic preprocessing steps by default with commonly used default parameters. But it is *very important* to look at your data to see the effects of these preprocessings and validate them.

In addition, the `NiftiMasker` is a scikit-learn compliant transformer so that you can directly plug it into a scikit-learn pipeline. This feature can be seen in *nifti_masker_advanced*.

5.4.1 Niimg

Niimg (pronounce ni-image) is a common term used in Nisl. It can either represents:



- a file path to a Nifti image
- any object exposing `get_data()` and `get_affine()` methods (it is obviously intended to handle nibabel's Nifti1Image but also user custom types if needed).

The Nifti Masker requires a 4-dimensional Nifti-like data. Accepted inputs are:

- Path to a 4-dimensional Nifti image
- List of paths to 3-dimensional Nifti images
- 4-dimensional Nifti-like object
- List of 3-dimensional Nifti-like objects

Note: Image affines

If you provide a sequence of Nifti images, all of them must have the same affine.

5.4.2 Custom data loading

Sometimes, you may want to preprocess data by yourself. In this example, we will restrict Haxby dataset to 150 frames to speed up computation. To do that, we load the dataset, restrain it to 150 frames and build a brand new Nifti like object to give it to the Nifti masker. There is no need to save your data in a file to pass it to Nifti masker. Simply use your Niimg !

5.4.3 Custom Masking

In the basic tutorial, we showed how the masker could compute a mask automatically: the result was quite impressive. But, on some datasets, the default algorithm may perform poorly. That is why it is very important to *always look at what your data looks like*.

Mask Visualization

Before exploring the subject, we define an helper function to display the masks. This function will display a background (compose of a mean of epi scans) and the mask as a red layer over this background.

Computing the mask

As said before, if a mask is not given, the Nifti Masker will try to compute one. It is *very important* to take a look at the generated mask, to see if it is suitable for your data and adjust parameters if it is not. See documentation for a complete list of mask computation parameters.

As an example, we will now try to build a mask on a dataset form scratch. Haxby dataset will be used since it provides a mask that we can use as a reference.

The first the step of the generation is to generate a mask with default parameters and take a look at it. As an indicator, we can, for example, compare the mask to original data.

With naked eyes, we can see that there is a problem with this mask. In fact, it does not cover a part of the brain and the outline of the mask is not very smooth. As we want to enlarge the mask a little bit and make it smoother, we try to apply opening (*mask_opening=true*).

This is not very effective. If we look at `nisl.masking.compute_epi_mask` (page **??**) documentation, we spot two interesting parameters: *lower_cutoff* and *upper_cutoff*. The algorithm seems to ignore dark (low) values. Without getting into the details of the algorithm, this means that the threshold is chosen into high values. We can tell the algorithm to ignore high values by lowering *upper cutoff*. Default value is 0.9, so we try 0.8.

The resulting mask seems correct. If we compare it to the original one, they are very close.

5.4.4 Preprocessings

Resampling

Nifti Masker offers two ways to resample images:

- *target_affine*: resample (resize, rotate...) images by providing a new affine
- *target_shape*: resize images by providing directly a new shape

Resampling can be used for example to reduce processing time of an algorithm by lowering image resolution.

Temporal Filtering

All previous filters concern spatial filtering. On the time axis, the Nifti masker also proposes some filters.

By default, the signal will be normalized. If the dataset provides a confounds file, it can be applied by providing the path to the file to the masker. Low pass and High pass filters allows one to remove artefacts.

Detrending removes linear trend along axis from data. It is not activated by default in the Nifti Masker but it is almost essential.

Note: Exercise

You can, more as a training than as an exercise, try to play with the parameters in Nisl examples. Try to enable detrending in haxby decoding and run it: does it have a big impact on the results ?

5.4.5 Inverse transform: unmasking data

Once that your computation is finished, you want to unmask your data to be able to visualize it. This step is present in almost all the examples provided in Nisl.

5.5 Masking the data manually

5.5.1 Extracting a brain mask

If we do not have a mask of the relevant regions available, a brain mask can be easily extracted from the fMRI data using the `nisl.masking.compute_epi_mask` (page **??**) function:

`compute_epi_mask` (page **??**)(mean_epi[, lower_cutoff, ...])	Compute a brain mask from fMRI data in 3D or 4D ndarrays.

nisl.masking.compute_epi_mask

`nisl.masking.`**`compute_epi_mask`**(*mean_epi*, *lower_cutoff=0.2*, *upper_cutoff=0.9*, *connected=True*, *opening=True*, *exclude_zeros=False*, *ensure_finite=True*, *verbose=0*)

Compute a brain mask from fMRI data in 3D or 4D ndarrays.

This is based on an heuristic proposed by T.Nichols: find the least dense point of the histogram, between fractions lower_cutoff and upper_cutoff of the total image histogram.

In case of failure, it is usually advisable to increase lower_cutoff.

Parameters **mean_epi: 3D or 4D array or nifti like image** :

EPI image, used to compute the mask.

lower_cutoff : float, optional

lower fraction of the histogram to be discarded.

upper_cutoff: float, optional :

upper fraction of the histogram to be discarded.

connected: boolean, optional :

if connected is True, only the largest connect component is kept.

opening: boolean, optional :

if opening is True, an morphological opening is performed, to keep only large structures. This step is useful to remove parts of the skull that might have been included.

ensure_finite: boolean :

If ensure_finite is True, the non-finite values (NaNs and infs) found in the images will be replaced by zeros

exclude_zeros: boolean, optional :

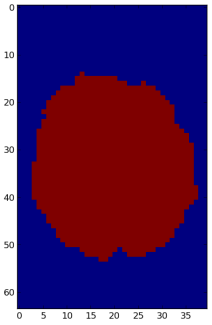
Consider zeros as missing values for the computation of the threshold. This option is useful if the images have been resliced with a large padding of zeros.

verbose: integer, optional :

Returns **mask** : 3D boolean ndarray

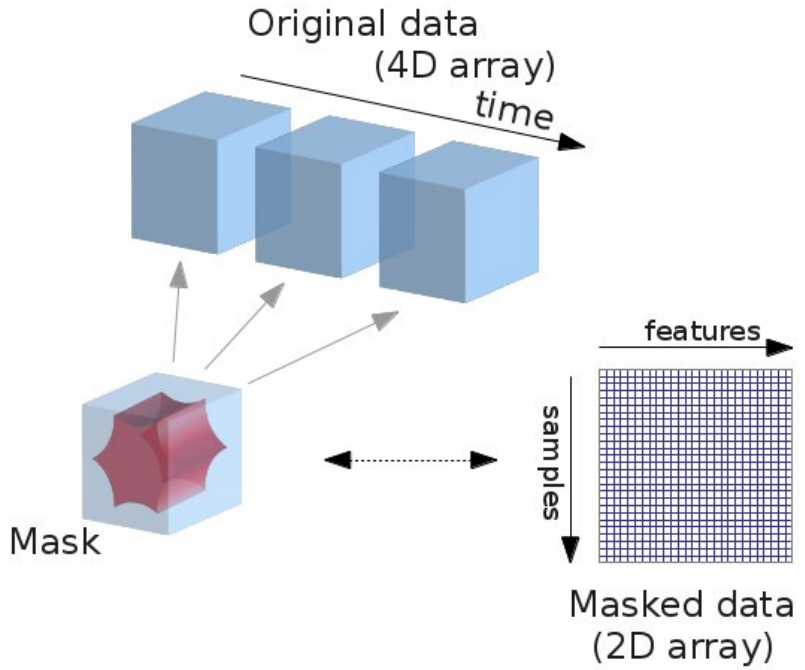
The brain mask

```
# Simple computation of a mask from the fMRI data
from nisl.masking import compute_epi_mask
mask = compute_epi_mask(mean_img)

# We create a new figure
pl.figure()
# A plot the axial view of the mask to compare with the axial
# view of the raw data displayed previously
pl.imshow(np.rot90(mask[:, :, 32]), interpolation='nearest')
```



5.5.2 From 4D to 2D arrays

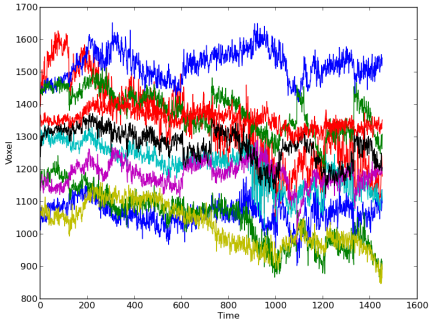
FMRI data is naturally represented as a 4D block of data: 3 spatial dimensions and time. In practice, we are most often only interested in working only on the time-series of the voxels in the brain. It is convenient to apply a brain mask and go from a 4D array to a 2D array, *voxel* **x** *time*, as depicted below:




```
# Applying the mask is just a simple array manipulation
masked_data = fmri_data[mask]

# masked_data is now a voxel x time matrix. We can plot the first 10
# lines: they correspond to time-series of 10 voxels on the side of the
# brain
pl.figure()
pl.plot(masked_data[:10].T)
pl.xlabel('Time')
pl.ylabel('Voxel')

pl.show()
```

5.6 Preprocessing functions

TODO



Reference

This is the class and function reference of nisl. Please refer to the *full user guide* for further details, as the class and function raw specifications may not be enough to give full guidelines on their uses.

List of modules

6.1 `nisl.datasets`: Automatic Dataset Fetching

Utilities to download NeuroImaging datasets

User guide: See the *datasets* section for further details.

6.1.1 Functions

`datasets.fetch_haxby` (page **??**)([data_dir, n_subjects, ...])	Download and loads complete haxby dataset
`datasets.fetch_haxby_simple` (page **??**)([data_dir, url, ...])	Download and loads an example haxby dataset
`datasets.fetch_nyu_rest` (page **??**)([n_subjects, ...])	Download and loads the NYU resting-state test-retest dataset
`datasets.fetch_adhd` (page **??**)([n_subjects, data_dir, ...])	Download and loads the ADHD resting-state dataset

nisl.datasets.fetch_haxby

nisl.datasets.**fetch_haxby**(*data_dir=None*, *n_subjects=1*, *url=None*, *resume=True*, *verbose=0*)
Download and loads complete haxby dataset

Parameters **data_dir: string, optional** :

Path of the data directory. Used to force data storage in a specified location. Default: None

n_subjects: integer, optional :

Number of subjects, from 1 to 5.

Returns **data: Bunch** :

Dictionary-like object, the interest attributes are : 'anat': string list

Paths to anatomic images

'func': string list Paths to nifti file with bold data

'session_target': string list Paths to text file containing session and target data

'mask_vt': string list Paths to nifti ventral temporal mask file

'mask_face': string list Paths to nifti ventral temporal mask file

'mask_house': string list Paths to nifti ventral temporal mask file

'mask_face_little': string list Paths to nifti ventral temporal mask file

'mask_house_little': string list Paths to nifti ventral temporal mask file

Notes

PyMVPA provides a tutorial using this dataset : http://www.pymvpa.org/tutorial.html

More informations about its structure : http://dev.pymvpa.org/datadb/haxby2001.html

See additional information[1]

References

Haxby, J., Gobbini, M., Furey, M., Ishai, A., Schouten, J., and Pietrini, P. (2001). Distributed and overlapping representations of faces and objects in ventral temporal cortex. Science 293, 2425-2430.

nisl.datasets.fetch_haxby_simple

nisl.datasets.**fetch_haxby_simple**(*data_dir=None*, *url=None*, *resume=True*, *verbose=0*)
Download and loads an example haxby dataset

Parameters **data_dir: string, optional** :

Path of the data directory. Used to force data storage in a specified location. Default: None

Returns **data: Bunch** :

[1] http://www.sciencemag.org/content/293/5539/2425

Dictionary-like object, the interest attributes are : 'func': string

Path to nifti file with bold data

'session_target': string Path to text file containing session and target data

'mask': string Path to nifti mask file

'session': string Path to text file containing labels (can be used for LeaveOneLabelOut cross validation for example)

Notes

PyMVPA provides a tutorial using this dataset : http://www.pymvpa.org/tutorial.html

More informations about its structure : http://dev.pymvpa.org/datadb/haxby2001.html

See additional information[2]

References

Haxby, J., Gobbini, M., Furey, M., Ishai, A., Schouten, J., and Pietrini, P. (2001). Distributed and overlapping representations of faces and objects in ventral temporal cortex. Science 293, 2425-2430.

nisl.datasets.fetch_nyu_rest

nisl.datasets.**fetch_nyu_rest**(*n_subjects=None, sessions=[1], data_dir=None, verbose=0*)
Download and loads the NYU resting-state test-retest dataset

Parameters **n_subjects: integer optional** :

The number of subjects to load. If None is given, all the subjects are used.

n_sessions: array of integers optional :

The sessions to load. Load only the first session by default.

data_dir: string, optional :

Path of the data directory. Used to force data storage in a specified location. Default: None

Returns **data** : Bunch

Dictionary-like object, the interest attributes are : 'func': string list

Paths to functional images

'anat_anon': string list Paths to anatomic images

'anat_skull': string Paths to skull-stripped images

'session': numpy array List of ids corresponding to images sessions

[2] http://www.sciencemag.org/content/293/5539/2425

Notes

This dataset is composed of 3 sessions of 26 participants (11 males). For each session, three sets of data are available:

•anatomical:

–anonymized data (defaced thanks to BIRN defacer)

–skullstripped data (using 3DSkullStrip from AFNI)

•functional

For each participant, 3 resting-state scans of 197 continuous EPI functional volumes were collected :

•39 slices

•matrix = 64 x 64

•acquisition voxel size = 3 x 3 x 3 mm

Sessions 2 and 3 were conducted in a single scan session, 45 min apart, and were 5-16 months after Scan 1.

All details about this dataset can be found here : http://cercor.oxfordjournals.org/content/19/10/2209.full

References

Documentation http://www.nitrc.org/docman/?group_id=274

Download http://www.nitrc.org/frs/?group_id=274

Paper to cite The Resting Brain: Unconstrained yet Reliable[3] Z. Shehzad, A.M.C. Kelly, P.T. Reiss, D.G. Gee, K. Gotimer, L.Q. Uddin, S.H. Lee, D.S. Margulies, A.K. Roy, B.B. Biswal, E. Petkova, F.X. Castellanos and M.P. Milham.

Other references

- The oscillating brain: Complex and Reliable[4] X-N. Zuo, A. Di Martino, C. Kelly, Z. Shehzad, D.G. Gee, D.F. Klein, F.X. Castellanos, B.B. Biswal, M.P. Milham
- Reliable intrinsic connectivity networks: Test-retest evaluation using ICA and dual regression approach[5], X-N. Zuo, C. Kelly, J.S. Adelstein, D.F. Klein, F.X. Castellanos, M.P. Milham

nisl.datasets.fetch_adhd

`nisl.datasets.`**`fetch_adhd`**(*n_subjects=None*, *data_dir=None*, *url=None*, *resume=True*, *verbose=0*)

Download and loads the ADHD resting-state dataset

Parameters **n_subjects: integer optional** :

The number of subjects to load. If None is given, all the 40 subjects are used.

data_dir: string, optional :

Path of the data directory. Used to force data storage in a specified location. Default: None

url: string, optional :

[3] http://cercor.oxfordjournals.org/content/19/10/2209
[4] http://dx.doi.org/10.1016/j.neuroimage.2009.09.037
[5] http://dx.doi.org/10.1016/j.neuroimage.2009.10.080

Override download URL. Used for test only (or if you setup a mirror of the data).

Returns **data** : Bunch

Dictionary-like object, the interest attributes are : 'func': string list

Paths to functional images

'parameters': string list Parameters of preprocessing steps

References

Download

ftp://www.nitrc.org/fcon_1000/htdocs/indi/adhd200/sites/ ADHD200_40sub_preprocessed.tgz

6.2 nisl.utils: Manipulating Niimgs

Validation and conversion utilities.

User guide: See the *utils* section for further details.

6.2.1 Functions

utils.is_a_niimg (page **??**)(object)	Check for get_data and get_affine method in an object
utils.check_niimg (page **??**)(niimg)	Check that an object is a niimg and load it if necessary
utils.check_niimgs (page **??**)(niimgs[, accept_3d])	Check that an object is a list of niimg and load it if necessary
utils.concat_niimgs (page **??**)(niimgs)	Concatenate a list of niimgs

nisl.utils.is_a_niimg

`nisl.utils.`**`is_a_niimg`**(*object*)

Check for get_data and get_affine method in an object

Parameters **object: unknown object** :

Tested object

Returns **True if get_data and get_affine methods are present and callable,** :

False otherwise. :

nisl.utils.check_niimg

`nisl.utils.`**`check_niimg`**(*niimg*)

Check that an object is a niimg and load it if necessary

Parameters **niimg: string or object** :

If niimg is a string, consider it as a path to Nifti image and call nibabel.load on it. If it is an object, check if get_data and get_affine methods are present, raise an Exception otherwise.

Returns **A nifti-like object (for the moment, nibabel.Nifti1Image)** :

Notes

In Nisl, special care has been taken to make image manipulation easy. This method is a kind of pre-requisite for any data processing method in Nisl as it check if data has the right format and load it if necessary.

Its application is idempotent.

nisl.utils.check_niimgs

nisl.utils.**check_niimgs**(*niimgs*, *accept_3d=False*)

Check that an object is a list of niimg and load it if necessary

Parameters **niimgs: (list of)* string or object** :

If niimgs is a list, checks if data is really 4D. Then, considering that it is a list of niimg and load them one by one. If niimg is a string, consider it as a path to Nifti image and call nibabel.load on it. If it is an object, check if get_data and get_affine methods are present, raise an Exception otherwise.

Returns **A list of nifti-like object (for the moment, nibabel.Nifti1Image)** :

Notes

This application is the pendant of check_niimg for niimages with a session level.

Its application is idempotent.

nisl.utils.concat_niimgs

nisl.utils.**concat_niimgs**(*niimgs*)

Concatenate a list of niimgs

Parameters **niimgs: array of niimgs** :

List of niimgs to concatenate. Can be paths to Nifti files or numpy matrices.

Returns **A single niimg** :

6.3 nisl.masking: Data Masking Utilities

Utilities to compute a brain mask from EPI images

User guide: See the *Custom Masking* (page **??**) section for further details.

6.3.1 Functions

masking.compute_epi_mask (page **??**)(mean_epi[, ...])	Compute a brain mask from fMRI data in 3D or 4D ndarrays.
masking.compute_multi_epi_mask (page **??**)(session_epi)	Compute a common mask for several sessions or subjects of fM
masking.intersect_masks (page **??**)(input_masks[, ...])	Compute intersection of several masks
masking.apply_mask (page **??**)(niimgs, mask_img[, ...])	Extract time series using specified mask
masking.unmask (page **??**)(X, mask[, transpose])	Take masked data and bring them back into 3D

nisl.masking.compute_epi_mask

nisl.masking.**compute_epi_mask**(*mean_epi*, *lower_cutoff=0.2*, *upper_cutoff=0.9*, *connected=True*, *opening=True*, *exclude_zeros=False*, *ensure_finite=True*, *verbose=0*)

Compute a brain mask from fMRI data in 3D or 4D ndarrays.

This is based on an heuristic proposed by T.Nichols: find the least dense point of the histogram, between fractions lower_cutoff and upper_cutoff of the total image histogram.

In case of failure, it is usually advisable to increase lower_cutoff.

Parameters **mean_epi: 3D or 4D array or nifti like image** :

EPI image, used to compute the mask.

lower_cutoff : float, optional

lower fraction of the histogram to be discarded.

upper_cutoff: float, optional :

upper fraction of the histogram to be discarded.

connected: boolean, optional :

if connected is True, only the largest connect component is kept.

opening: boolean, optional :

if opening is True, an morphological opening is performed, to keep only large structures. This step is useful to remove parts of the skull that might have been included.

ensure_finite: boolean :

If ensure_finite is True, the non-finite values (NaNs and infs) found in the images will be replaced by zeros

exclude_zeros: boolean, optional :

Consider zeros as missing values for the computation of the threshold. This option is useful if the images have been resliced with a large padding of zeros.

verbose: integer, optional :

Returns **mask** : 3D boolean ndarray

The brain mask

nisl.masking.compute_multi_epi_mask

nisl.masking.**compute_multi_epi_mask**(*session_epi*, *lower_cutoff=0.2*, *upper_cutoff=0.9*, *connected=True*, *opening=True*, *threshold=0.5*, *exclude_zeros=False*, *n_jobs=1*, *verbose=0*)

Compute a common mask for several sessions or subjects of fMRI data.

Uses the mask-finding algorithms to extract masks for each session or subject, and then keep only the main connected component of the a given fraction of the intersection of all the masks.

Parameters **session_files: list 3D or 4D array** :

A list arrays, each item is a subject or a session.

threshold: float, optional :

the inter-session threshold: the fraction of the total number of session in for which a voxel must be in the mask to be kept in the common mask. threshold=1 corresponds to keeping the intersection of all masks, whereas threshold=0 is the union of all masks.

lower_cutoff: float, optional :

lower fraction of the histogram to be discarded.

upper_cutoff: float, optional :

upper fraction of the histogram to be discarded.

connected: boolean, optional :

if connected is True, only the largest connect component is kept.

exclude_zeros: boolean, optional :

Consider zeros as missing values for the computation of the threshold. This option is useful if the images have been resliced with a large padding of zeros.

n_jobs: integer, optional :

The number of CPUs to use to do the computation. -1 means 'all CPUs'.

Returns **mask** : 3D boolean ndarray

The brain mask

nisl.masking.intersect_masks

nisl.masking.**intersect_masks**(*input_masks*, *threshold=0.5*, *connected=True*)

Compute intersection of several masks

Given a list of input mask images, generate the output image which is the the threshold-level intersection of the inputs

Parameters **input_masks: list of ndarrays** :

3D individual masks

threshold: float within [0, 1[, optional :

gives the level of the intersection. threshold=1 corresponds to keeping the intersection of all masks, whereas threshold=0 is the union of all masks.

connected: bool, optional :

If true, extract the main connected component

Returns **grp_mask, boolean array of shape the image shape** :

nisl.masking.apply_mask

nisl.masking.**apply_mask**(*niimgs*, *mask_img*, *dtype=<type 'numpy.float32'>*, *smooth=False*, *ensure_finite=True*, *transpose=False*)

Extract time series using specified mask

Read the time series from the given nifti images or filepaths, using the mask.

Parameters **niimgs: list 4D (ot list of 3D) nifti images (or filenames)** :

Images to be masked.

mask: 3d ndarray :

3D mask array: true where a voxel should be used.

smooth: False or float, optional :

If smooth is not False, it gives the size, in voxel of the spatial smoothing to apply to the signal.

ensure_finite: boolean :

If ensure_finite is True, the non-finite values (NaNs and infs) found in the images will be replaced by zeros

transpose: boolean, optional :

Indicate if data must be transposed after masking.

Returns **session_series: ndarray** :

2D array of time series (voxel, time)

Notes

When using smoothing, ensure_finite should be True: as elsewhere non finite values will spread accross the image.

nisl.masking.unmask

nisl.masking.**unmask**(*X*, *mask*, *transpose=False*)

Take masked data and bring them back into 3D

This function is intelligent and will process data of any dimensions. It iterates until data has only one dimension and then it tries to unmask it. An error is raised if masked data has not the right number of voxels.

Parameters **X: (list of)* numpy array** :

Masked data. You can provide data of any dimension so if you want to unmask several images at one time, it is possible to give a list of images.

mask: numpy array of boolean values :

Mask of the data

transpose: boolean, optional :

Indicates if data must be transposed after unmasking.

Returns **data: (list of)* 3D numpy array** :

Unmasked data: 1D or 2D arrays are converted into 3D or 4D arrays resp. The number of dimensions is respected wrt input data.

6.4 nisl.resampling: Data Resampling Utilities

Utilities to resample a Nifti Image

User guide: See the *Resampling* (page **??**) section for further details.

6.4.1 Functions

resampling.to_matrix_vector (page ??)(transform)	Split a transform into it's matrix and vector components.
resampling.from_matrix_vector (page ??)(matrix, vector)	Combine a matrix and vector into a homogeneous transform.
resampling.get_bounds (page ??)(shape, affine)	Return the world-space bounds occupied by an array given an
resampling.resample_img (page ??)(niimg[, ...])	Resample a Nifti Image

nisl.resampling.to_matrix_vector

nisl.resampling.**to_matrix_vector**(*transform*)

Split a transform into it's matrix and vector components.

The tranformation must be represented in homogeneous coordinates and is split into it's rotation matrix and translation vector components.

Parameters **transform** : ndarray

Transform matrix in homogeneous coordinates. Example, a 4x4 transform representing rotations and translations in 3 dimensions.

Returns **matrix, vector** : ndarray

The matrix and vector components of the transform matrix. For an NxN transform, matrix will be N-1xN-1 and vector will be 1xN-1.

See Also:

from_matrix_vector (page ??)

nisl.resampling.from_matrix_vector

nisl.resampling.**from_matrix_vector**(*matrix*, *vector*)

Combine a matrix and vector into a homogeneous transform.

Combine a rotation matrix and translation vector into a transform in homogeneous coordinates.

Parameters **matrix** : ndarray

An NxN array representing the rotation matrix.

vector : ndarray

A 1xN array representing the translation.

Returns **xform** : ndarray

An N+1xN+1 transform matrix.

See Also:

to_matrix_vector (page ??)

nisl.resampling.get_bounds

nisl.resampling.**get_bounds**(*shape*, *affine*)

Return the world-space bounds occupied by an array given an affine.

nisl.resampling.resample_img

nisl.resampling.**resample_img**(*niimg*, *target_affine=None*, *target_shape=None*, *interpolation='continuous'*, *copy=True*)

Resample a Nifti Image

Parameters **niimg: nisl nifti image** :

Path to a nifti file or nifti-like object

target_affine: numpy matrix, optional :

If specified, the image is resampled corresponding to this new affine. target_affine can be a 3x3 or a 4x4 matrix

target_shape: 3-tuple, optional :

If specified, the image will be resized to match this new shape.

interpolation: string, optional :

Can be continuous' (default) or 'nearest'. Indicate the resample method

copy: boolean, optional :

If true, copy source data to avoid side-effects.

6.5 nisl.signals: Preprocessing Time Series

Preprocessing functions for time series.

User guide: See the *signals* section for further details.

6.5.1 Functions

signals.clean (page **??**)(signals[, confounds, ...])	Normalize the signal, and if any confounds are given, project in the orthogor

nisl.signals.clean

nisl.signals.**clean**(*signals*, *confounds=None*, *low_pass=0.2*, *t_r=2.5*, *high_pass=False*, *detrend=False*, *standardize=True*, *shift_confounds=False*)

Normalize the signal, and if any confounds are given, project in the orthogonal space.

Low pass filter improves specificity (more interesting arrows selected)

High pass filter should be kepts small, so as not to kill sensitivity

6.6 nisl.io: Loading and Preprocessing files easily

The nisl.io (page **??**) module includes scikit-learn tranformers and tools to preprocess neuro-imaging data.

User guide: See the *The NiftiMasker: loading, masking and filtering* (page **??**) and *nifti_masker_advanced* section for further details.

6.6.1 Classes

nifti_masker.NiftiMasker (page **??**)([sessions, mask, ...])	Nifti data loader with preprocessing
nifti_multi_masker.NiftiMultiMasker (page **??**)([mask, ...])	Nifti data loader with preprocessing for multiple subjects

nisl.io.nifti_masker.NiftiMasker

class nisl.io.nifti_masker.**NiftiMasker**(*sessions=None*, *mask=None*, *mask_connected=True*, *mask_opening=False*, *mask_lower_cutoff=0.2*, *mask_upper_cutoff=0.9*, *smooth=False*, *standardize=False*, *detrend=False*, *target_affine=None*, *target_shape=None*, *low_pass=None*, *high_pass=None*, *t_r=None*, *transpose=False*, *memory=Memory(cachedir=None)*, *transform_memory=Memory(cachedir=None)*, *verbose=0*)

Nifti data loader with preprocessing

Parameters **mask: filename or NiImage, optional** :

Mask of the data. If not given, a mask is computed in the fit step. Optional parameters detailed below (mask_connected...) can be set to fine tune the mask extraction.

sessions: numpy array, optional :

Add a session level to the preprocessing. Each session will be detrended independently. Must be a 1D array of n_samples elements.

smooth: False or float, optional :

If smooth is not False, it gives the size, in voxel of the spatial smoothing to apply to the signal.

standardize: boolean, optional :

If standardize is True, the time-series are centered and normed: their mean is put to 0 and their variance to 1.

detrend: boolean, optional :

This parameter is passed to signals.clean. Please see the related documentation for details

low_pass: False or float, optional :

This parameter is passed to signals.clean. Please see the related documentation for details

high_pass: False or float, optional :

This parameter is passed to signals.clean. Please see the related documentation for details

t_r: float, optional :

This parameter is passed to signals.clean. Please see the related documentation for details

memory: instance of joblib.Memory or string :

Used to cache the masking process. By default, no caching is done. If a string is given, it is the path to the caching directory.

transform_memory: instance of joblib.Memory or string :

Used to cache the perprocessing step. By default, no caching is done. If a string is given, it is the path to the caching directory.

verbose: interger, optional :

Indicate the level of verbosity. By default, nothing is printed

target_affine: 3x3 or 4x4 matrix, optional :

This parameter is passed to resampling.resample_img. Please see the related documentation for details.

target_shape: 3-tuple of integers, optional :

This parameter is passed to resampling.resample_img. Please see the related documentation for details.

mask_connected: boolean, optional :

If mask is None, this parameter is passed to masking.compute_epi_mask for mask computation. Please see the related documentation for details.

mask_opening: boolean, optional :

If mask is None, this parameter is passed to masking.compute_epi_mask for mask computation. Please see the related documentation for details.

mask_lower_cutoff: float, optional :

If mask is None, this parameter is passed to masking.compute_epi_mask for mask computation. Please see the related documentation for details.

mask_upper_cutoff: float, optional :

If mask is None, this parameter is passed to masking.compute_epi_mask for mask computation. Please see the related documentation for details.

transpose: boolean, optional :

If True, data is transposed after preprocessing step.

See Also:

`nisl.masking.compute_epi_mask` (page **??**), `nisl.resampling.resample_img` (page **??**), `nisl.masking.apply_mask` (page **??**), `nisl.signals.clean` (page **??**)

Attributes

Methods

`fit` (page **??**)(niimgs[, y])	Compute the mask corresponding to the data
`fit_transform` (page **??**)(X[, y, confounds])	Fit to data, then transform it
`get_params` (page **??**)([deep])	Get parameters for the estimator
`inverse_transform`(X)	
`set_params` (page **??**)(**params)	Set the parameters of the estimator.
`transform` (page **??**)(niimgs[, confounds])	Apply mask, spatial and temporal preprocessing
`transform_single_niimgs`(niimgs[, sessions, ...])	

__init__(*sessions=None*, *mask=None*, *mask_connected=True*, *mask_opening=False*, *mask_lower_cutoff=0.2*, *mask_upper_cutoff=0.9*, *smooth=False*, *standardize=False*, *detrend=False*, *target_affine=None*, *target_shape=None*, *low_pass=None*, *high_pass=None*, *t_r=None*, *transpose=False*, *memory=Memory(cachedir=None)*, *transform_memory=Memory(cachedir=None)*, *verbose=0*)

fit(*niimgs*, *y=None*)
Compute the mask corresponding to the data

Parameters **niimgs: list of filenames or NiImages** :

Data on which the mask must be calculated. If this is a list, the affine is considered the same for all.

fit_transform(*X*, *y=None*, *confounds=None*, ***fit_params*)
Fit to data, then transform it

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters **X** : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

Returns **X_new** : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params(*deep=True*)
Get parameters for the estimator

Parameters **deep: boolean, optional** :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

set_params(***params*)
Set the parameters of the estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns **self** :

transform(*niimgs*, *confounds=None*)
Apply mask, spatial and temporal preprocessing

Parameters **niimgs: nifti like images** :

Data to be preprocessed

confounds: CSV file path or 2D matrix :

This parameter is passed to signals.clean. Please see the related documentation for details

nisl.io.nifti_multi_masker.NiftiMultiMasker

class nisl.io.nifti_multi_masker.**NiftiMultiMasker**(*mask=None*, *mask_connected=True*, *mask_opening=False*, *mask_lower_cutoff=0.2*, *mask_upper_cutoff=0.9*, *smooth=False*, *standardize=False*, *detrend=False*, *target_affine=None*, *target_shape=None*, *low_pass=None*, *high_pass=None*, *t_r=None*, *transpose=False*, *n_jobs=1*, *memory=Memory(cachedir=None)*, *transform_memory=Memory(cachedir=None)*, *verbose=0*)

Nifti data loader with preprocessing for multiple subjects

Parameters **mask: filename or NiImage, optional** :

Mask of the data. If not given, a mask is computed in the fit step. Optional parameters detailed below (mask_connected...) can be set to fine tune the mask extraction.

smooth: False or float, optional :

If smooth is not False, it gives the size, in voxel of the spatial smoothing to apply to the signal.

standardize: boolean, optional :

If standardize is True, the time-series are centered and normed: their mean is put to 0 and their variance to 1.

detrend: boolean, optional :

This parameter is passed to signals.clean. Please see the related documentation for details

low_pass: False or float, optional :

This parameter is passed to signals.clean. Please see the related documentation for details

high_pass: False or float, optional :

This parameter is passed to signals.clean. Please see the related documentation for details

t_r: float, optional :

This parameter is passed to signals.clean. Please see the related documentation for details

memory: instance of joblib.Memory or string :

Used to cache the masking process. By default, no caching is done. If a string is given, it is the path to the caching directory.

transform_memory: instance of joblib.Memory or string :

Used to cache the perprocessing step. By default, no caching is done. If a string is given, it is the path to the caching directory.

n_jobs: integer, optional :

The number of CPUs to use to do the computation. -1 means 'all CPUs'.

verbose: interger, optional :

Indicate the level of verbosity. By default, nothing is printed

target_affine: 3x3 or 4x4 matrix, optional :

This parameter is passed to resampling.resample_img. Please see the related documentation for details.

target_shape: 3-tuple of integers, optional :

This parameter is passed to resampling.resample_img. Please see the related documentation for details.

mask_connected: boolean, optional :

If mask is None, this parameter is passed to masking.compute_epi_mask for mask computation. Please see the related documentation for details.

mask_opening: boolean, optional :

If mask is None, this parameter is passed to masking.compute_epi_mask for mask computation. Please see the related documentation for details.

mask_lower_cutoff: float, optional :

If mask is None, this parameter is passed to masking.compute_epi_mask for mask computation. Please see the related documentation for details.

mask_upper_cutoff: float, optional :

If mask is None, this parameter is passed to masking.compute_epi_mask for mask computation. Please see the related documentation for details.

transpose: boolean, optional :

If True, data is transposed after preprocessing step.

See Also:

nisl.masking.compute_epi_mask (page **??**), nisl.resampling.resample_img (page **??**), nisl.masking.apply_mask (page **??**), nisl.signals.clean (page **??**)

Attributes

Methods

fit (page **??**)(niimgs[, y])	Compute the mask corresponding to the data
fit_transform (page **??**)(X[, y, confounds])	Fit to data, then transform it
get_params (page **??**)([deep])	Get parameters for the estimator
inverse_transform(X)	
set_params (page **??**)(**params)	Set the parameters of the estimator.
transform (page **??**)(niimgs[, confounds])	Apply mask, spatial and temporal preprocessing
transform_single_niimgs(niimgs[, sessions, ...])	

__init__(*mask=None*, *mask_connected=True*, *mask_opening=False*, *mask_lower_cutoff=0.2*, *mask_upper_cutoff=0.9*, *smooth=False*, *standardize=False*, *detrend=False*, *target_affine=None*, *target_shape=None*, *low_pass=None*, *high_pass=None*, *t_r=None*, *transpose=False*, *n_jobs=1*, *memory=Memory(cachedir=None)*, *transform_memory=Memory(cachedir=None)*, *verbose=0*)

fit (*niimgs*, *y=None*)
Compute the mask corresponding to the data

Parameters **niimgs: list of filenames or NiImages** :

Data on which the mask must be calculated. If this is a list, the affine is considered the same for all.

fit_transform (*X*, *y=None*, *confounds=None*, ***fit_params*)
Fit to data, then transform it

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

Parameters **X** : numpy array of shape [n_samples, n_features]

Training set.

y : numpy array of shape [n_samples]

Target values.

Returns **X_new** : numpy array of shape [n_samples, n_features_new]

Transformed array.

get_params (*deep=True*)
Get parameters for the estimator

Parameters **deep: boolean, optional** :

If True, will return the parameters for this estimator and contained subobjects that are estimators.

set_params (***params*)
Set the parameters of the estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The former have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns **self** :

transform (*niimgs*, *confounds=None*)
Apply mask, spatial and temporal preprocessing

Parameters **niimgs: nifti like images** :

Data to be preprocessed

confounds: CSV file path or 2D matrix :

This parameter is passed to signals.clean. Please see the related documentation for details