NeuroImaging with scikit-learn

Release 0.1

Gaël Varoquaux and Alexandre Abraham

http://nisl.github.com

July 02, 2012

Contents

Machine Learning in NeuroImaging: what and why

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.

1.2 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 <http://www.springerlink.com/content/7377x70p5515v778/>]'_**).

[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

CHAPTER 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

2.1 Installation of the materials useful for this tutorial

2.1.1 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[1] or PythonXY[2]: 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[3] 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[4] 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
- ipython

[1]http://www.enthought.com/products/epd.php
[2]http://code.google.com/p/pythonxy/
[3]http://www.enthought.com/products/epd.php
[4]http://www.enthought.com/products/epd.php

2.1.2 Nibabel

Nibabel[5] 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
```

2.1.3 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
```

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[6].

We will be using IPython[7], 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
```

2.2.1 Scientific computing

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

Numerical arrays

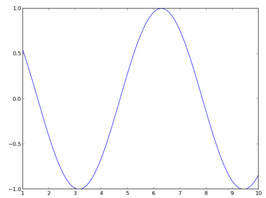
```
>>> import numpy as np
>>> t = np.linspace(1, 10, 2000)  # 2000 points between 1 and 10
>>> t
array([  1.        ,   1.00450225,   1.0090045 , ...,   9.9909955 ,
         9.99549775,  10.        ])
>>> t / 2
array([ 0.5       ,  0.50225113,  0.50450225, ...,  4.99549775,
        4.99774887,  5.        ])
```

[5] http://nipy.sourceforge.net/nibabel/
[6] http://scipy-lectures.github.com/
[7] http://ipython.org


```
>>> 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...[8]

Plotting




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

More documentation...[9]

Image processing

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

More documentation...[10]

Signal processing

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

More documentation...[11]

Much more

- Simple statistics:

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

- Linear algebra:

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

More documentation...[12]

[8] http://scipy-lectures.github.com/intro/numpy/index.html
[9] http://scipy-lectures.github.com/intro/matplotlib/matplotlib.html
[10] http://scipy-lectures.github.com/advanced/image_processing/index.html
[11] http://scipy-lectures.github.com/intro/scipy.html#signal-processing-scipy-signal
[12] http://scipy-lectures.github.com/intro/scipy.html

2.2.2 Scikit-learn: machine learning

The core concept in the scikit-learn[13] is the estimator object, for instance an SVC (support vector classifier[14]). 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[15].

2.3 Finding help

Reference material

- A quick and gentle introduction to scientific computing with Python can be found in the scipy lecture notes[16].

[13] http://scikit-learn.org
[14] http://scikit-learn.org/stable/modules/svm.html
[15] http://scikit-learn.org/stable/tutorial/index.html
[16] http://scipy-lectures.github.com/

- 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 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

CHAPTER 3

Basic dataset manipulation: loading and visualisation

3.1 Downloading the tutorial data

This tutorial package embeds tools to download and load datasets. They can be imported from `nisl.datasets`:

```
>>> from nisl import datasets
>>> haxby_data = datasets.fetch_haxby()
>>> # The data is then already loaded as numpy arrays:
>>> haxby_data.keys()
['files', 'session', 'target', 'target_strings', 'data', 'affine', 'mask']
>>> haxby_data.data.shape # 1452 time points and a spatial size of 40x64x64
(40, 64, 64, 1452)
```

`fetch_haxby` (page 8)([data_dir])	Download and loads the haxby dataset
`fetch_nyu_rest` (page 9)([n_subjects, data_dir])	Download and loads the NYU resting-state test-retest dataset

3.1.1 nisl.datasets.fetch_haxby

`nisl.datasets.`**`fetch_haxby`**(*data_dir=None*)

Download and loads the 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 : 'data' : numpy array : the data to learn 'target' : numpy array

target of the data

'mask' : the masks for the data 'session' : the labels for LeaveOneLabelOut cross validation

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.

3.1.2 nisl.datasets.fetch_nyu_rest

`nisl.datasets.`**`fetch_nyu_rest`**(*n_subjects=None*, *data_dir=None*)

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.

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 : 'data' : numpy array : the data to learn 'target' : numpy array

target of the data

'mask' : the masks for the data 'xyz' : index to 3D-coordinate array

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

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

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[2] 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[3] 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[4], X-N. Zuo, C. Kelly, J.S. Adelstein, D.F. Klein, F.X. Castellanos, M.P. Milham

The data are downloaded only once and stored locally in the *nisl_data* folder. Note that you can copy that folder across computers to avoid downloading.

3.2 Loading Nifti or analyze files

NIfTI and Analyse files

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[5]. Once the file is downloaded, a single line is needed to load it.

```
from nisl import datasets
haxby = datasets.fetch_haxby()

# Get the file names relative to this dataset
files = haxby.files
bold = files[1]

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

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

Dataset formatting: data shape

We can find two main representations for MRI scans:

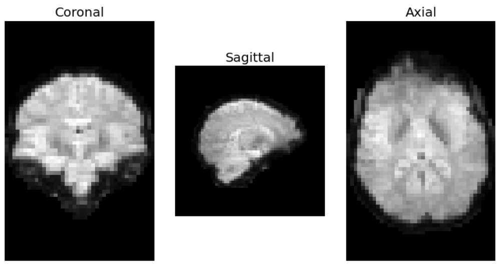
- a big 4D matrix representing 3D MRI along time, stored in a big 4D NifTi file. FSL[a] users tend to prefer this format.
- several 3D matrices representing each volume (time point) of the session, stored in set of 3D Nifti or analyse files. SPM[b] users tend to prefer this format.

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

3.3 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)
```



3.4 Masking the data

3.4.1 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_mask` (page 12) function:

`compute_mask` (page 12)(epi_img[, lower_cutoff, ...])	Compute a brain mask from fMRI data in 3D or 4D ndarrays.

nisl.masking.compute_mask

nisl.masking.**compute_mask**(*epi_img*, *lower_cutoff=0.2*, *upper_cutoff=0.9*, *connected=True*, *exclude_zeros=False*)

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 **epi_img** : 3D ndarray

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.

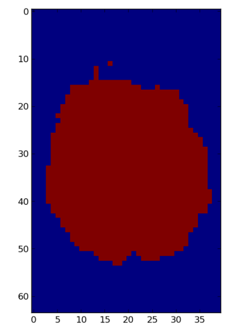
exclude_zeros: boolean, optional :

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.

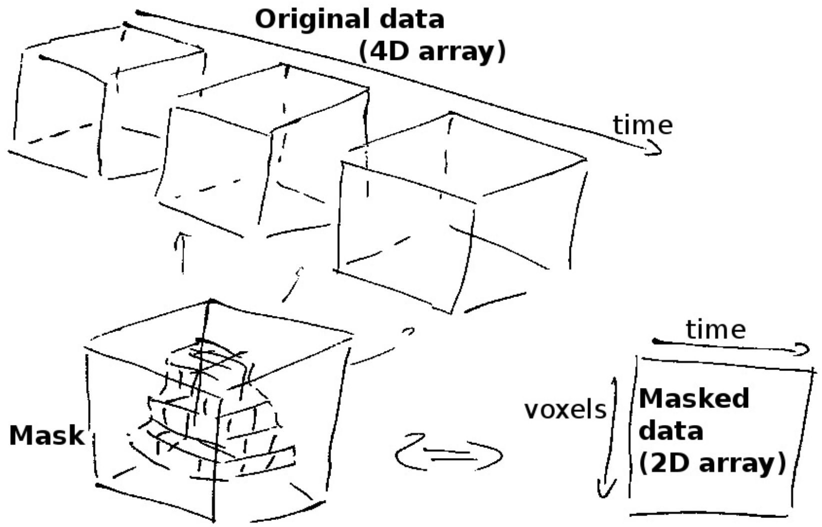
Returns **mask** : 3D boolean ndarray

The brain mask

```
# Simple computation of a mask from the fMRI data
from nisl.masking import compute_mask
```

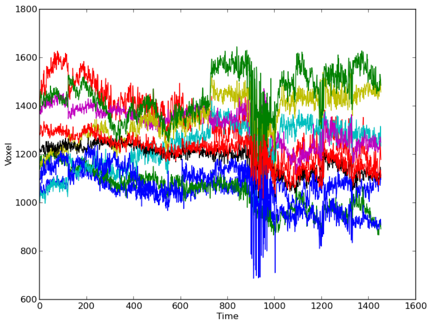



go from a 4D array to a 2D array, *voxel* **x** *time*, as depicted below:




```
# 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()
```



CHAPTER 4

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.

4.1 Decoding on simulated data

Objectives

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

4.1.1 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 $\mathbf{X}$:

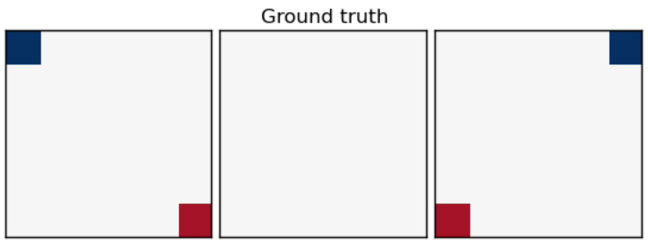
$$\mathbf{y} = \mathbf{X}\mathbf{w} + \mathbf{e}$$

- $\mathbf{w}$: 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* (page **??**):

```
X_train, X_test, y_train, y_test, snr, noise, coefs, size =\
        create_simulation_data(snr=10, n_samples=400, size=12)


coefs = np.reshape(coefs, [size, size, size])
```

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




```
plot_slices(coefs, title="Ground truth")

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

4.1.2 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)),
]
```

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.

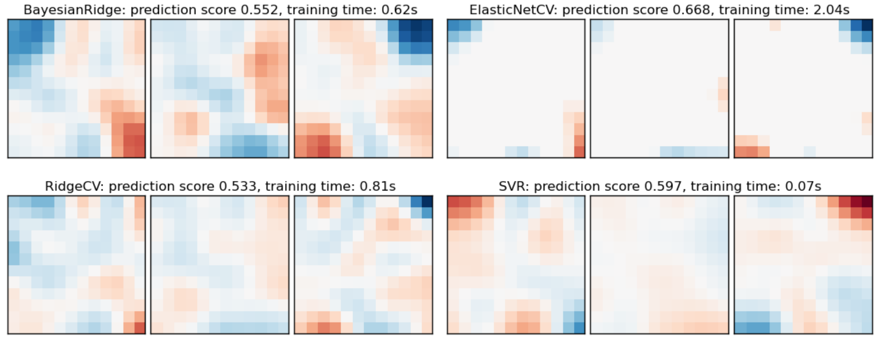
As the estimators expose a fairly consistent API, we can all fit them in a for loop: they all have a *fit* method for fitting the data, a *score* method to retrieve the prediction score, and because they are all linear models, a *coef_* attribute that stores the coefficients **w** estimated.

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
    coefs = classifier.coef_
```

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


```
    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)

    # 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.

4.2 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.

4.2.1 Data loading and preprocessing

We launch ipython:

```
$ ipython -pylab
```

First, we load the data using the tutorial's data downloader, `nisl.datasets.fetch_haxby` (page 8):

```
from nisl import datasets
dataset = datasets.fetch_haxby()
fmri_data = dataset.data
mask = dataset.mask
affine = dataset.affine
y = dataset.target
conditions = dataset.target_strings
session = dataset.session

# 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 12) for a discussion on using masks)
- finally detrend the data for each session

```
import numpy as np

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

# Process the data in order to have a two-dimensional design matrix X of
# shape (n_samples, n_features).
X = fmri_data[mask].T

# X.shape is (n_samples, n_features): (1452, 39912)

# Detrend data on each session independently
from scipy import signal
for s in np.unique(session):
    X[session == s] = signal.detrend(X[session == s], axis=0)
```

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 = X[condition_mask]
y = y[condition_mask]
```



```
session = session[condition_mask]
conditions = conditions[condition_mask]

# We now have n_samples, n_features = X.shape = 864, 39912
n_samples, n_features = X.shape

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

4.2.2 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.

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

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

```
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=1000)

# 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)
```

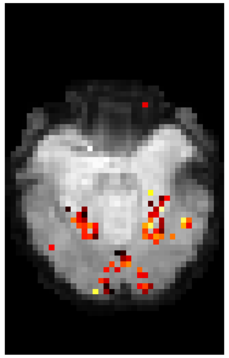
Warning ! Do not do this at home: the prediction that we obtain here is to good to be true (see next paragraph). Here we are just doing a sanity check.

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 previousely-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
act = np.zeros(mean_img.shape)
act[mask != 0] = svc[0]

# We use a masked array so that the voxels at '0' are displayed
# transparently
act = np.ma.masked_array(act, act == 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,
          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.Nifti1Image(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, which
# corresponds to a leave-one-session-out
cv = LeaveOneLabelOut(session)

### 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, verbose=10)
```

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* (page **??**). Now, all you have to do is to publish the results :)

4.2.3 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].

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:

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


```
>>> 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...

4.3 Searchlight : finding voxels containing maximum information

4.3.1 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.

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

4.3.2 Preprocessing

Loading

As seen in *previous sections* (page 8), fetching the data from internet and loading it can be done with the provided functions:

```
from nisl import datasets
dataset = datasets.fetch_haxby()
fmri_data = dataset.data
mask = dataset.mask
affine = dataset.affine
y = dataset.target
conditions = dataset.target_strings
session = dataset.session
```

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
- detrend the data

```
import numpy as np

# Change axis in order to have X under n_samples * x * y * z
X = np.rollaxis(fmri_data, 3)
# X.shape is (1452, 40, 64, 64)

# Mean image: used as background in visualisation
mean_img = np.mean(X, axis=0)

# Detrend data on each session independently
from scipy import signal
for s in np.unique(session):
    X[session == s] = signal.detrend(X[session == s], axis=0)
```

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.


```
mask = (dataset.mask != 0)
process_mask = mask.copy()
process_mask[..., 38:] = False
process_mask[..., :36] = False
process_mask[:, 30:] = False
```

Restricting the dataset

Like in the *decoding* (page 17) example, 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 = X[condition_mask]
y = y[condition_mask]
session = session[condition_mask]
conditions = conditions[condition_mask]
```

4.3.3 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)
```

4.3.4 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.

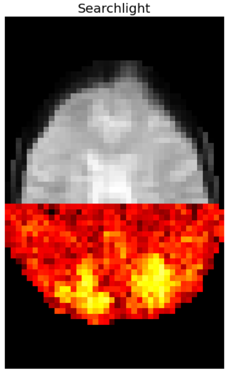
[13] http://scikit-learn.org/supervised_learning.html
[14] http://scikit-learn.org/supervised_learning.html

```
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, y)
```

4.3.5 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.

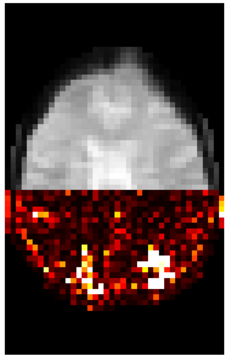



```
import pylab as pl
pl.figure(1)
# searchlight.scores_ contains per voxel cross validation scores
s_scores = np.ma.array(searchlight.scores_, mask=np.logical_not(process_mask))
pl.imshow(np.rot90(mean_img[..., 37]), interpolation='nearest',
        cmap=pl.cm.gray)
pl.imshow(np.rot90(s_scores[...,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

[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.




```
from sklearn.feature_selection import f_classif
pl.figure(2)
X_masked = X[:, 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()
```

CHAPTER 5

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).

5.1 fMRI clustering

5.1.1 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 9).

5.1.2 Preprocessing

Loading

As seen in *previous sections* (page 8), we fetch the data from internet and load it with a provided function:

```
from nisl import datasets
dataset = datasets.fetch_nyu_rest(n_subjects=1)
```

Masking

No mask is given with the data so we have to compute one ourselves, using the function `nisl.masking.compute_mask` (page 12):

```
fmri_data = dataset.func[0]

# Compute a brain mask
from nisl import masking
mask = masking.compute_mask(fmri_data)
```

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


```
# Mask data: go from a 4D dataset to a 2D dataset with only the voxels
# in the mask
fmri_masked = fmri_data[mask]
```

The result is a numpy array of boolean that is used to mask our original *X*.

5.1.3 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 500 clusters and we time it.

```
from sklearn.cluster import WardAgglomeration
import time
start = time.time()
ward = WardAgglomeration(n_clusters=500, connectivity=connectivity,
                         memory='nisl_cache')
ward.fit(fmri_masked.T)
print "Ward agglomeration 500 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 1000 elements.

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

```
# the caching mechanism
start = time.time()
ward = WardAgglomeration(n_clusters=1000, connectivity=connectivity,
                         memory='nisl_cache')
ward.fit(fmri_masked.T)
print "Ward agglomeration 1000 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.

5.1.4 Post-Processing and visualization

Unmasking

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

```
import numpy as np
labels = - np.ones(mask.shape)
labels[mask] = ward.labels_
```

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

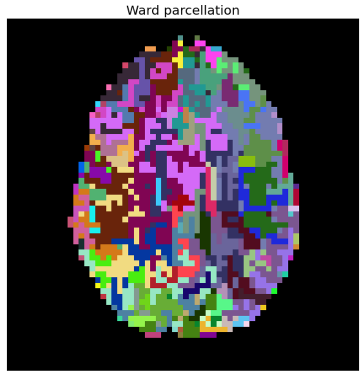
We can visualize the clusters. We assign random colors to each cluster for the labels visualization.

```
import pylab as pl

# Cut at z=20
cut = labels[:, :, 20].astype(np.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
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')
```

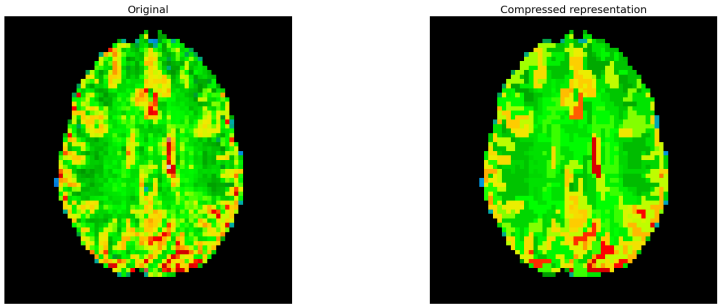
Compressed picture

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_fmri_img = fmri_data[..., 0].copy()
# Outside the mask: a uniform value, smaller than inside the mask
first_fmri_img[np.logical_not(mask)] = 0.9*first_fmri_img[mask].min()
vmax = first_fmri_img[..., 20].max()
pl.imshow(np.rot90(first_fmri_img[..., 20]),
          interpolation='nearest', cmap=pl.cm.spectral, vmax=vmax)
pl.axis('off')
pl.title('Original')

# 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.T)

# Display the corresponding data compressed using the parcellation
fmri_compressed = ward.inverse_transform(fmri_reduced)
compressed_img = first_fmri_img.copy()
compressed_img[mask] = fmri_compressed[0]

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



We can see that using only 1000 parcels, we can approximate well the original image.

5.2 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

5.2.1 Preprocessing

Loading

As seen in *previous sections* (page 8), 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=3)
```

Conctenating, smoothing and masking

```
# Concatenate all the subjects
fmri_data = np.concatenate(dataset.func, axis=3)

# Apply a small amount of Gaussian smoothing: in the case of ICA it is
# important as it introduces a spatial model that ICA lacks and greatly
# reduces the high-frequency signal
from scipy import ndimage
for image in fmri_data.T:
    # This works efficiently because image is a view on fmri_data
    image[...] = ndimage.gaussian_filter(image, 1.5)

# Take the mean along axis 3: the direction of time
mean_img = np.mean(fmri_data, axis=3)
```



```
# Mask non brain areas
from nisl import masking
mask = masking.compute_mask(mean_img)
data_masked = fmri_data[mask]
```

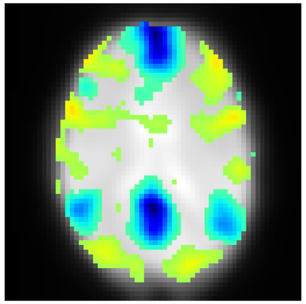
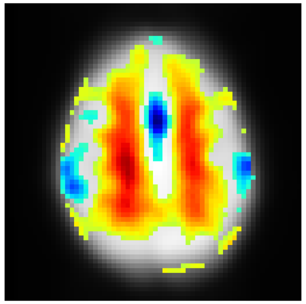
5.2.2 Applying ICA

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

# 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) < .5] = 0

# Now we inverting the masking operation, to go back to a full 3D
# representation
(x, y, z) = mean_img.shape
components = np.zeros((x, y, z, n_components))
components[mask] = components_masked

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

5.2.3 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
import pylab as pl
pl.figure()
pl.axis('off')
vmax = np.max(np.abs(components[:, :, 20, 16]))
pl.imshow(np.rot90(mean_img[:, :, 20]), interpolation='nearest',
          cmap=pl.cm.gray)
pl.imshow(np.rot90(components[:, :, 20, 16]), 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_img[:, :, 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.