CS9.432.S22: Cognitive Science and AI

Assignment 1 - Basic understanding and masking of brain images with Nilearn

Maximum marks - 40

Instructions for submission

Deadline for submission on 25th January 2022 Before 12midnight

- You may do the assignment in Jupyter or Colab notebook. The output of each *TODO* should be clearly visible within the notebook.
- You need to submit a notebook specified by Roll Number roll_no.ipynb in Moodle before deadline.
- Include the assignment number, your name and roll number in the notebook as well for better identity.
- No late submissions are accepted.
- **IMPORTANT**: Make sure that the assignment that you submit is your own work. Do not copy any part from any source including your friends, seniors or the internet. Any breach of this rule could result in serious actions including an F grade in the course.
- Your grade will depend on the correctness of answers and output. In addition, due consideration will be given to the clarity and details of your answers and the legibility and structure of your code.
- Unless specifically permitted, collaborations are not allowed.
- Do not copy or plagiarise, if you're caught for plagiarism or copying, penalties are much higher (including an F grade in the course) than simply omitting that question.

Objective

To understand, manipulate and transform Neuroimaging data for practicing Machine Learning or Deep Learning. This could be an opportunity to introduce Nilearn - a Python package that will serve our objectives. Nilearn is a open-source software with good documentation, plenty of examples in the gallery and easy-to-use.

Nilearn provides tools for analysis techniques like functional connectivity, multivariate (machine-learning based) "decoding", but also more "basic" tools like image manipulation and visualization. From this assignment, we will learn how to use the basic tools from Nilearn for downloading and processing Magnetic Resonance Imaging (MRI) data.

The installation of the Nilearn is pretty straight forward following install nilearn documentation.

If you want to read an entire functionality that Nilearn offers? Go through their online website or check out this article.

Note

This assignment expects basic familiarity with Python. Most of the content is inspired from Nilearn documentation. Credits to Nilearn team!

1. Understanding Neuroimaging data

A brief background on clinical imaging data formats and what needs to be done for processing with Nilearn: from the scanner to computer.

From the MRI scanner, the images are collected in a raw scanner format called as DICOM (Digital Imaging and Communications in Medicine) and needs to be converted to NIfTI (Neuroimaging Informatics Technology Initiative) format to make working with the data easier. Typically, one can convert DICOM data to NIfTI using dcm2niix. Learning about this conversion is not the objective of this assignment. Hence, we skip those details. (Optional) The background information about NIfTI can be found here.

NIfTI/Nifti data

Nifti images are identified as "image.nii" or the compressed version of it as "image.nii.gz". Nilearn works with MRI data stored in the Nifti images via the dependency on Nibabel package. Many functionalities in Nilearn accept either filenames (the path to a nifti file) or a Nibabel Nifti1Image object.

3D/4D Nifti images

MRI data is acquired as 3D or 4D images.

3D image (x, y, z) is a single timepoint anatomical or T1-weighted image which is high in spatial resolution and differentiates easily between brain tissues. One 3D image also called as brain volume has the information encoded in a voxels similar to pixels used to display the images. Each voxel is cubic and has certain dimensions. These T1 weighted images are used to study the brain anatomy (i.e., shape, volume, cortical thickness, brain region). Each slice (2D) can be extracted from this 3D image/volume.

Note

The difference between dimension of cubic voxel and dimension of an image. Both are not same. This will be cleared while loading the images and looking at the loaded image properties like shape, affine, header, etc. After the coding exercise this might become clear. Don't worry for now.

A 4D image (x, y, z + time) or multiple 3D volumes in time namely functional Magnetic Resonance Image (fMRI), that captures the Blood Oxygenation Level Dependent (BOLD) signal changes in the brain or simply brain activity. Each such 4D image is acquired using T2* weighted imaging sequence and has poor spatial resolution compared to anatomical acquisition. If one voxel is selected from a fMRI image, this voxel represents the BOLD signal changes in time. This information encoded in each voxel is crucial for studying localization of activity in the brain in response to the external stimuli.

Now, Let's download the example data from the internet with Nilearn.

Downloading MRI data is pretty straight forward with Nilearn as it has entire module called as *nilearn.datasets* written to fetch example data such as anatomical, functional or atlases. Let's import it below and fetch famous Haxby experimental dataset that includes: anatomical, functional, masks/ROIs, experimental details in csv. For now, we explore anatomical, functional and masks. Experimental data in csv is necessary when we do decoding or machine learning.

```python
# This might take a while, depending on your internet speed
from nilearn import datasets
haxby_dataset = datasets.fetch_haxby(data_dir=None)
# by default this function returns images of second subject from the Haxby study

# ignore warnings or if you don't want warnings, uncomment and run below code
# import warnings
# warnings.simplefilter("ignore")
```

They return a data structure that contains different pieces of information. We can explore them on the fly. Before, we explore where the data is downloaded and stored.

nilearn_data: Where is the downloaded data stored?

The fetching functions download the reference datasets to the disk. They save it locally for future use, in one of the following directories:

- the folder specified by *data_dir* parameter in the fetching function
- the *nilearn_data* folder in the user home folder

You can check via a function a default directory nilearn will store the data with *nilearn.datasets.get_data_dirs*.

```
print(datasets.get_data_dirs())
```

Now, Let's inspect in more details the attributes of the downloaded Haxby data.

```
# The different pieces of information
print(sorted(list(haxby_dataset.keys())))
```

Read the *"description"* in more detail, which describes the contents of the dictionary:

```
print(haxby_dataset['description'].decode("utf-8"))
```

Alright, having read the description identify the anatomical image and functional image.

```
# TODO 1: print anatomical and functional image to locate their paths
print(haxby_dataset[?]) # insert anatomical image key here
print(haxby_dataset[?]) # insert functional image key here
```

Food for thought: Are they path to Nifti images or nibabel loaded Nifti1Image?

Note

By default, Nilearn returns paths as a list. The most basic functions in Nilearn requires a 3D image rather than the list.

Food for thought: How many subjects images are seen in a list? What is the name of that subject? Hint: The name of the subject can be seen from that returned list. You could also try to download another subject if you would like.

Load the images

Alright, now we have some data to work with. With the *image* module in Nilearn, we can load in and perform many different operations on nifti images. We'll import it below:

```
from nilearn import image
```

Now, let's load the anatomical image and functional image and check whether loaded image are still a path to Nifti image or loaded nibabel Nifti1Image.

```
# TODO 2: Load anatomical image and functional image
anatomical = image.load_img(img=?) # specify the relevant input here
type(anatomical)
functional = image.load_img(img=?) # specify the functional image
type(functional)
```

The difference between path to Nifti images (not loaded in-memory) and Nifti1Image (loaded in-memory).

> ***Note***
>
> Nilearn's *image.load_img* is basically the same as the *nibabel.load* function, but with some extra functionality (like loading in a list of files using wildcards) and checks (like whether it's really a nifti image).

Now, examine the shape of the anatomical and functional image. The shape, affine, header, etc are the parameters of the Nifti1Image. We can only see them when loaded as nibabel Nifti1Image.

```
anatomical_image_shape = anatomical.shape
functional_image_shape = functional.shape
assert(anatomical_image_shape == (124, 256, 256, 1))
assert(functional_image_shape == (40, 64, 64, 1452))
```

Food for thought: Functional MRI image has how many time points acquired?

The fourth dimension is the time dimension. As can be seen from the above snippet that anatomical image has single time dimension (one brain volume 3D) whereas functional MRI has multiple timepoints. These time series data are analysed for decoding and statistical analysis which we will cover in subsequent assignments.

Also, look at the *affine* parameter similarly to how shape is extracted. This helps to get an idea about Anatomy of NIfTI. Additional hands on understanding of Nifti parameters can be found here Anatomy_of_Nifti.

In the "haxby2001/subj2" subdirectory of the download directory which can be found using *datasets.get_data_dirs()*, there are multiple nifti files with names starting with "mask". These masks outlines the Regions of Interest (ROIs) for this particular subject. Can you load them all at once using the *image.load_img* function with a wildcard or globbing? Assign the result in a variable named all_mask_images, which should be a 4D Nifti1Image object.

```
# TODO 3: Wildcard like path should be specificied to load all mask images at once
all_mask_images = image.load_img(?)
assert(all_mask_images.shape == (40, 64, 64, 5))
```

Food for thought: How many mask images are loaded into one Nifti1Image object?

Data Visualization

Nilearn package is popularly known for its visualization 3D and 4D brain images. Let's try to visualize 3D brain images using module *nilearn.plotting*. Under the hood Nilearn uses *matplotlib* as a dependency for visualization. You don't have to learn Matplotlib but could be an advantage for advanced usage.

```
from nilearn import plotting
```

Simply plot anatomical image

```
plotting.plot_anat(anatomical)
plotting.show() # equivalent to %matplotlib inline in Jupyter or Colab Notebook
```

This displays the anatomical image of subj2 in x, y, z display directions. Inspect the parameter *display_mode* to choose between one or two directions.

Try! more cuts in 3D volume to slices and one display direction. These display directions are also called as axial, coronal and sagittal.

```
plotting.plot_anat(anatomical, display_mode='x', cut_coords=[-40, -20, 0, 20, 40])
```

Similarly, try plotting any one mask image using *plotting.plot_roi* below. Hint: the mask images can be found be found in the retrieved data either the variable assigned while downloading them or path to download directory.

```
# TODO 4: Specify the mask image input to plot ROI images
plotting.plot_roi(roi_img=?)
```

More_plotting_tools

2. Image manipulation

Average functional image over time (compute mean image given 4D)

Nilearn provides some functions to make your life easier when doing array mathematics on 3D or 4D images. For example, to compute the mean of the functional image we can use *image.mean_img* to compute the mean across time for every voxel in a 4D image.

> ***Note***
>
> This mean calculation can be done using NumPy given that the images are converted to NumPy data matrices. As NumPy is not designed to take directly the Nifti images. Such NumPy operations can be easily done directly on the Nifti images with Nilearn.

```
# Since we already imported image module from Nilearn, we can use it again here
# TODO 5: Specify functional image as input as it is 4D and we want to average in time
mean_functional_image = image.mean_img(imgs=?)

assert(mean_functional_image.shape == (40, 64, 64))

# TODO 6: Visualize the mean image using `plotting.plot_epi`
plotting.plot_epi(epi_img=?)
plotting.show()
```

Food for thought: EPI stands for Echo Planar Imaging a way of acquiring fMRI - scanner specifications. We don't have to go deeper into it.

Smooth an image

Similarly, we can also do simple Gaussian smoothing using *image.smooth_img* Can we smooth the mean functional image that is computed above?

```
# TODO 7: Specify the mean functional image
smooth_mean_functional_image = image.smooth_img(img=?, fwhm=8)

# Visualize the smoothed image using `plotting.plot_epi`
# TODO 8: Specify the input
plotting.plot_epi(epi_img=?)
```

Mathematical tricks applied directly onto the Nifti images

We can even do complex mathematical operations by calling directly the numpy related operations onto the Nifti images without a necessity to convert to the numpy understandable arrays.

```
log_image = image.math_img("np.log(img)", img=anatomical)

# negate the image

negate_image = image.math_img("-img", img=anatomical)
```

Similarly, simply add the logarithm image computed above to anatomical image.

```
# TODO 9: Use addition symbol with two images
add_anatomical_img_with_log_img = image.math_img(?)
```

Compute the voxelwise TSNR (mean across time divided by standard deviation across time for each voxel) of the functional image using *image.math_img* and assign a variable to save the computed tsnr image. Then, plot the image using plot_epi. Hint: use np.mean and np.std inspired by seeing how log image is computed.

```
# TODO 10: Use Numpy functions
tsnr_functional_image = image.math_img(?)
plotting.plot_epi(tsnr_functional_image)
plotting.show()
```

Changing affine to look similar across images (simply Resampling images)

As we can see that the shape of anatomical and functional image are different. We can make them into the same shape using another image manipulation function *image.resample_to_img*. Image to image resampling.

Resample anatomical (source) to functional image (target) using *image.resample_to_img* and print the shape of the output image. This should display that the affines are same for both images.

```
# Implement here
# TODO 11: Follow the documentation of resample_to_img
```

More information about Resampling

3. Masking brain images for easier Machine Learning

A common operation in fMRI analyses is masking: extracting particular voxels from the entire dataset, usually based on a binary brain mask. Masking, at least in fMRI analyses, is often done on the spatial dimensions of 4D images; as such, masking can be seen as a operation that takes in a 4D image with spatial dimensions *X*Y*Z* and temporal dimension *T* and returns a *T*K* 2D array, where *K* is the number of voxels that falls under the masking procedure.

Reasons to mask your data could be, for example, to exclude non-brain voxels (like in skullstripping) or to perform confirmatory region-of-interest (ROI) analyses.

Nilearn provides several functions and classes that perform masking, which differ in how extensive they are (some only perform masking on a single image, others do this for multiple images at the same time, and/or may additionally perform preprocessing steps). Importantly, all take in a 4D niimg-like object and return a 2D numpy array.

We'll first take a look at the most simple and low-level implementation: apply_mask. This function takes in a 4D image (which will be masked), a binary 3D image (i.e., with only zeros and ones, where ones indicate that they should be included) as mask. Let's do this for our data (functional image) using the brain mask computed on the functional image itself:

```python
from nilearn import masking

# Let's compute the epi mask
func_mask = masking.compute_epi_mask(mean_functional_image)

# Visualize the mask that is computed on the mean functional image

plotting.plot_epi(mean_functional_image)
plotting.plot_roi(func_mask, mean_functional_image)
plotting.show() # equivalent to %matplotlib inline in Jupyter or Colab Notebook
```

Visualizing the mask is always useful as a quality check. For instance we can check whether the mask strictly overlaps with the images to be masked.

Now, mask the functional image using *nilearn.masking.apply_mask*

```python
print("Before masking, our data has shape %s ..." % (functional.shape,))
functional_masked = masking.apply_mask(functional, func_mask)
print("After masking, shape %s and is a %s" % (functional_masked.shape, type(functional_masked).__name__))
```

You can see that after masking output shape is a 2D matrix. We have transformed 4D to 2D which we could input this matrix to decoding model.

Now, repeat the same analysis as above but use face mask that is provided with the Haxby study. This mask should be called as haxby_dataset['mask_face'] or any other masks that are provided in data download folder.

```python
# TODO 12:
# Complete the block with face mask or any other mask by following above demo
```

Finally, a masking can be done using a high-level object implemented in module *nilearn.input_data.NiftiMasker*. This object that has some extra preprocessing features. Unlike the name suggests, this class does much more than masking: it also (optionally) allows you to spatially and temporally preprocess your data! It works slightly differently than the relatively simple functions shown so far though. We will start understanding it more carefully.

We'll start with importing it:

```python
from nilearn.input_data import NiftiMasker
```

Importantly, NiftiMasker is not a function, but a (custom) class. Read through the documentation of the NiftiMasker class to see which arguments it accepts.

For now, we'll initialize a very simple NiftiMasker that only accepts a particular brain mask (and set verbose=True to print some extra information).

```
masker = NiftiMasker(mask_img=func_mask, verbose=True)
```

We have now initialized the masker, you can also type simply masker in the console. Now use *masker.fit_transform* on functional image specified by a mask that was computed previously.

```
masked_functional_image = masker.fit_transform(functional)
print(masked_functional_image.shape)
```

Names can be assigned as per your convenience. Just make sure they are not overwritten. *fit* does simple checks on the provided mask and input functional images then *transform* extracts the apply_mask strategy i.e., extract 4D to 2D ndarray matrix.

> ***Note***
>
> The output shape using low-level or high-level object is same. The advantage of using high-level object is to apply several operations at one-go. A simple exercise below explains this advantage.

Having read the documentation of NiftiMasker, implement spatial smoothing fwhm=7 and use mask strategy='epi'. Save the output to a variable named as something like smoothed_masked_functional_image.

```
# TODO 13:
# Implement the smoothing and mask strategy parameter here, inspired by above
```

NiftiMasker does much more than smoothing. For example, one can do band pass filtering, confound regression and normalizing the data (mean 0 and standard deviation to 1). This object is easy to implement and with one line of code you can convert complex neuroimaging representations (4D) to brain representations (2D) that can be understood by Machine Learning models.

```
# TODO 14: Until now we have looked at steps that lead to 2D from 4D. Now,
# Implement a simple masking pipeline:
# 1. Download the fMRI data (4D), a new subject from the Haxby dataset.
# E.g., if "subj2 is used already then download "subj1" or "subj3".
# 2. Implement NiftiMasker, specify mask image while initialization of object
# and apply it on the fMRI data
# 3. Plot the mask image overlayed onto mean of fMRI data
# 4. Print the masked output, it should be 2D matrix
```

This assignment covered the basics of Nilearn, transforming 4D to 2D data matrices and other visualization of brain images. This will get you started for subsequent assignments.

Happy Hacking!