pyradigm Documentation

Release 0.4.1

Pradeep Reddy Raamana

Nov 05, 2017

Contents:

Pyradigm is a PYthon based data structure to ease and improve Dataset's InteGrity in Machine learning workflows.

CHAPTER 1

About

1.1 Background

A common problem for machine learning developers is keeping track of the source of the features extracted, and to ensure integrity of the dataset (e.g. not getting data mixed up from different subjects and/or classes). This is incredibly hard as the number of projects grow, or personnel changes are frequent. These aspects can break the chain of hyper-local info about various datasets, such as where did the original data come from, how was it processed or quality controlled, how was it put together, by who and what does some columns in the table mean etc. This package aims to provide a Python data structure to encapsulate a machine learning dataset with key info greatly suited for neuroimaging applications (or similar domains), where each sample needs to be uniquely identified with a subject ID (or something similar). Key-level correspondence across data, labels (e.g. 1 or 2), classnames (e.g. 'healthy', 'disease') and the related attributes helps maintain data integrity. Moreover, attributes like free-text description help annotate all the important information. The class methods offer the ability to arbitrarilty combine and subset datasets, while automatically updating their description reduces burden to keep track of the original source of features.

Check the *Usage examples* and *API Reference* pages, and let me know your comments.

1.2 Context

For users of Pandas, some of the elements in *pyradigm*'s API/interface may look familiar. However, the aim of this data structure is not to offer an alternative to pandas, but to ease the machine learning workflow for neuroscientists by

1. offering several well-knit methods and useful attributes specifically geared towards neuroscience research,
2. aiming to offer utilities that combines multiple or advanced patterns of routine dataset handling and
3. using a more accessible language (compared to hard to read pandas docs aimed at econometric audience) to better cater to neuroscience developers (esp. the novice).

Thanks for checking out. Your feedback will be appreciated.

CHAPTER 2

Installation

Pyradigm can easily be installed with a single command:

```
pip install pyradigm
```

If you lack `sudo` access, try

```
pip install pyradigm --user
```

2.1 Requirements

- Packages: `numpy`
- Supported versions: 2.7, 3.5 and 3.6

CHAPTER 3

Usage examples

This class is greatly suited for neuroimaging applications (or any other domain), where each sample needs to be uniquely identified with a subject ID (or something similar).

Key-level correspondence across data, labels (1 or 2), classnames ('healthy', 'disease') and the related helps maintain data integrity and improve the provenance, in addition to enabling traceback to original sources from where the features have been originally derived.

Just to given you a concrete examples, let's look at how an ML dataset is handled traditionally.

You have a matrix X of size n x p, with n samples and p features, and a vector y containing the target values (or class labels or class identifiers). This X and y serves as training (and test set) for a classifier like SVM to fit the data X to match y as accurately as possible.

Let's get a little more concrete:

```
import sys, os
import numpy as np
import matplotlib
%matplotlib
%matplotlib inline
import matplotlib.pyplot as plt

n = 10 # number of samples
p = 3  # number of features

X = np.random.random([n, p]) # random data for illustration
y = [1]*5 + [2]*5            # random labels ...

np.set_printoptions(precision=2) # save some screen space
print('X : \n{}'.format(X))
print('y : \n{}'.format(y))
```

```
Using matplotlib backend: TkAgg
X :
[[ 0.73  0.85  0.3 ]
```

```
 [ 0.63  0.09  0.87]
 [ 0.14  0.71  0.19]
 [ 0.25  0.33  0.08]
 [ 0.8   0.85  0.99]
 [ 0.78  0.76  0.47]
 [ 0.25  0.54  0.18]
 [ 0.57  0.98  0.36]
 [ 0.1   0.1   0.74]
 [ 0.16  0.76  0.53]]
y :
[1, 1, 1, 1, 1, 2, 2, 2, 2, 2]
```

Almost all the machine learning toolboxes take their input in this form: X and y, regardless of the original source that produced these features in the first place.

This is all fine if all you ever wanted to do is to extract some features, do some machine learning and dispose these features away!

** But this is almost never the case!**

Because it doesn't simply end there.

At a minimum, I often need to know * which samples are misclassified - meaning you need to know what the identifiers are and not simply their row indices in X? * what are the charecteristics of those samples? * what classes do they belong to?

And all this info needs to be obtained * without having to write lots of code connecting few non-obvious links to disparate sources of data (numerical features X, and sample identifiers in a CSV file) to find the relevant info * without having to track down who or which method originally produced these features * how the previous personnel or grad student organized the whole dataset, if you haven't generated the features yourself from scratch

And if you are like me, you would be thinking about how would you organize your workflow such that the aforementioned tasks can be accomplished with ease.

This data structure attempts to accomplish that with ease. By always organizing the extracted features keyed-in into a dictionary with their *sample id*, and other important info such as *target values* and other identified info. This, by definition, preserves the integrity of the data (inability to incorrectly label samples etc).

No, this data structure doesn't offer the full provenance tracking, which is quite a challenging problem. But it tries make your life a little easier in your ML workflows.

An example application is shown below, touching upon the following topics:

- Motivation
- Constructing a dataset
- Attributes
- Accessing samples
- Iteration over samples
- Subset selection
- Saving/reloading a dataset (Serialization)
- Combining datasets and diving them into useful subsets
- Portability (e.g. with sklearn)

Improting the necessary modules and our fancy class definition:

```
from pyradigm import MLDataset
```

We can now instantiate it and give it a description:

```
dataset = MLDataset()
dataset.description = 'ADNI1 baseline: cortical thickness features from Freesurfer v4.
↪3, QCed.'
```

```
dataset
```

```
ADNI1 baseline: cortical thickness features from Freesurfer v4.3, QCed.
Empty dataset.
```

You can see the dataset some description attached to it, however we know it is empty. This can be verified in a boolean context as shown below:

```
bool(dataset)
```

```
False
```

Let's add samples to this dataset which is when this dataset implementation becomes really handy. Before we do that, we will define some convenience routines defined to just illustrate a simple yet common use of this dataset.

```
def read_thickness(path):
    """Dummy function to minic a data reader."""

    # in your actural routine, this might be:
    #   pysurfer.read_thickness(path).values()
    return np.random.random(2)


def get_features(work_dir, subj_id):
    """Returns the whole brain cortical thickness for a given subject ID."""

    # extension to identify the data file; this could be .curv, anything else you␣
↪choose
    ext_thickness = '.thickness'

    thickness = dict()
    for hemi in ['lh', 'rh']:
        path_thickness = os.path.join(work_dir, subj_id, hemi + ext_thickness)
        thickness[hemi] = read_thickness(path_thickness)

    # concatenating them to build a whole brain feature set
    thickness_wb = np.concatenate([thickness['lh'], thickness['rh']])

    return thickness_wb
```

So now we have IO routines to read the data for us. Let's define where the data will come from:

```
work_dir = '/project/ADNI/FreesurferThickness_v4p3'
class_set = ['Cntrl', 'Alzmr', 'MCI']
class_sizes = [15, 12, 18]
```

This would obviously change for your applications, but this has sufficient properties to illustrate the point.

Let's look at what methods this dataset offers us:

```
dir(dataset)
```

```
['add_classes',
 'add_sample',
 'class_set',
 'class_sizes',
 'classes',
 'data',
 'data_and_labels',
 'del_sample',
 'description',
 'extend',
 'feature_names',
 'get_class',
 'get_feature_subset',
 'get_subset',
 'glance',
 'keys',
 'num_classes',
 'num_features',
 'num_samples',
 'random_subset',
 'random_subset_ids',
 'random_subset_ids_by_count',
 'sample_ids',
 'sample_ids_in_class',
 'save',
 'summarize_classes',
 'train_test_split_ids',
 'transform']
```

That's a lot of methods of convenience to organize and retrieve dataset.

So let's go through them by their usage sections.

CHAPTER 4

Constructor

You can see there few methods such as `add_sample`, `get_subset` etc: important method being `add_sample`, which is key to constructing this dataset. Let's go ahead and some samples:

To contruct a dataset, one typically starts with a list of subject IDs to be added - we create few random lists, each to be considered as a separate class:

```
import random
from datetime import datetime
random.seed(datetime.now())

def read_target_list(class_name, class_size):
    "Generates a random target list. In reality, you would do something like the␣
↪commented code below."
    target_list = list()
    for idx in range(class_size):
        target_list.append('{}{:04d}'.format(class_name[0],np.random.randint(1000)))

    return target_list
```

Now we go through each of the above classes, and add each sample that class to the dataset.

```
for class_index, class_id in enumerate(class_set):
    print('Working on class {:>5}'.format(class_id))

    target_list = read_target_list(class_id,class_sizes[class_index])
    for subj_id in target_list:
        print('\t reading subject {:>15}'.format(subj_id))
        thickness_wb = get_features(work_dir, subj_id)

        # adding the sample to the dataset
        dataset.add_sample(subj_id, thickness_wb, class_index, class_id)
```

```
Working on class Cntrl
     reading subject           C0562
     reading subject           C0408
```

```
     reading subject           C0760
     reading subject           C0170
     reading subject           C0241
     reading subject           C0980
     reading subject           C0822
     reading subject           C0565
     reading subject           C0949
     reading subject           C0041
     reading subject           C0372
     reading subject           C0141
     reading subject           C0492
     reading subject           C0064
     reading subject           C0557
Working on class Alzmr
     reading subject           A0034
     reading subject           A0768
     reading subject           A0240
     reading subject           A0042
     reading subject           A0141
     reading subject           A0888
     reading subject           A0032
     reading subject           A0596
     reading subject           A0969
     reading subject           A0215
     reading subject           A0074
     reading subject           A0229
Working on class   MCI
     reading subject           M0760
     reading subject           M0434
     reading subject           M0033
     reading subject           M0942
     reading subject           M0034
     reading subject           M0868
     reading subject           M0595
     reading subject           M0476
     reading subject           M0770
     reading subject           M0577
     reading subject           M0638
     reading subject           M0421
     reading subject           M0006
     reading subject           M0552
     reading subject           M0040
     reading subject           M0165
     reading subject           M0256
     reading subject           M0127
```

Nice. Isn't it?

So what's nice about this, you say? *The simple fact that you are constructing a dataset as you read the data* in its most elemental form (in the units of the dataset such as the subject ID in our neuroimaging application). You're done as soon as you're done reading the features from disk.

What's more - you can inspect the dataset in an intuitive manner, as shown below:

```
dataset
```

```
ADNI1 baseline: cortical thickness features from Freesurfer v4.3, QCed.
45 samples, 3 classes, 4 features.
```

```
Class Cntrl : 15 samples.
Class Alzmr : 12 samples.
Class   MCI : 18 samples.
```

Even better, right? No more coding of several commands to get the complete and concise sense of the dataset.

CHAPTER 5

Convenient attributes

If you would like, you can always get more specific information, such as:

```
dataset.num_samples
```

```
45
```

```
dataset.num_features
```

```
4
```

```
dataset.class_set
```

```
['MCI', 'Cntrl', 'Alzmr']
```

```
dataset.class_sizes
```

```
Counter({'Alzmr': 12, 'Cntrl': 15, 'MCI': 18})
```

```
dataset.class_sizes['Cntrl']
```

```
15
```

If you'd like to take a look data inside for few subjects - shall we call it a glance?

```
dataset.glance()
```

```
{'C0170': array([ 0.37,  0.78,  0.5 ,  0.79]),
 'C0241': array([ 0.11,  0.18,  0.58,  0.36]),
 'C0408': array([ 0.49,  0.38,  0.05,  0.82]),
 'C0562': array([ 0.64,  0.59,  0.01,  0.8 ]),
 'C0760': array([ 0.12,  0.51,  0.95,  0.23])}
```

We can control the number of items to glance, by passing a number to dataset.glance() method:

```
dataset.glance(2)
```

```
{'C0408': array([ 0.49,  0.38,  0.05,  0.82]),
 'C0562': array([ 0.64,  0.59,  0.01,  0.8 ])}
```

Or you may be wondering what are the subject IDs in the dataset.. here they are:

```
dataset.sample_ids
```

```
['C0562',
 'C0408',
 'C0760',
 'C0170',
 'C0241',
 'C0980',
 'C0822',
 'C0565',
 'C0949',
 'C0041',
 'C0372',
 'C0141',
 'C0492',
 'C0064',
 'C0557',
 'A0034',
 'A0768',
 'A0240',
 'A0042',
 'A0141',
 'A0888',
 'A0032',
 'A0596',
 'A0969',
 'A0215',
 'A0074',
 'A0229',
 'M0760',
 'M0434',
 'M0033',
 'M0942',
 'M0034',
 'M0868',
 'M0595',
 'M0476',
 'M0770',
 'M0577',
 'M0638',
 'M0421',
 'M0006',
 'M0552',
 'M0040',
 'M0165',
 'M0256',
 'M0127']
```

CHAPTER 6

Accessing samples

Thanks to elegant implementation, data for a given sample 'M0299' can simply be obtained by:

```
dataset['M0040']
```

```
array([ 0.27,  0.52,  0.61,  0.49])
```

Like a Python dict, it raises an error if the key is not in the dataset:

```
dataset['dlfjdjf']
```

```
---------------------------------------------------------------------------

KeyError                                  Traceback (most recent call last)

<ipython-input-22-4b19d52bac71> in <module>()
----> 1 dataset['dlfjdjf']


~/dev/pyradigm/pyradigm/pyradigm.py in __getitem__(self, item)
    839             return self.__data[item]
    840         else:
--> 841             raise KeyError('{} not found in dataset.'.format(item))
    842
    843     def __iter__(self):


KeyError: 'dlfjdjf not found in dataset.'
```

A more graceful handling would be to use `dataset.get` to control what value to be returned in case the requested id is not found in the dataset.

```
dataset.get('dkfjd', np.nan)
```

```
nan
```

CHAPTER 7

Iteration

Thanks to builtin iteration, we can easily iterate over all the samples:

```
for sample, features in dataset:
    print("{} : {:>10} : {}".format(sample, dataset.classes[sample], features))
```

```
C0562 :      Cntrl : [ 0.64  0.59  0.01  0.8 ]
C0408 :      Cntrl : [ 0.49  0.38  0.05  0.82]
C0760 :      Cntrl : [ 0.12  0.51  0.95  0.23]
C0170 :      Cntrl : [ 0.37  0.78  0.5   0.79]
C0241 :      Cntrl : [ 0.11  0.18  0.58  0.36]
C0980 :      Cntrl : [ 0.1   0.52  0.79  0.68]
C0822 :      Cntrl : [ 0.44  0.97  0.06  0.99]
C0565 :      Cntrl : [ 0.89  0.5   0.89  0.48]
C0949 :      Cntrl : [ 0.84  0.84  0.51  0.12]
C0041 :      Cntrl : [ 0.07  0.19  0.68  0.81]
C0372 :      Cntrl : [ 0.7   0.05  0.67  0.39]
C0141 :      Cntrl : [ 0.46  0.18  0.69  0.17]
C0492 :      Cntrl : [ 0.82  0.77  0.07  0.69]
C0064 :      Cntrl : [ 0.24  0.54  0.36  0.37]
C0557 :      Cntrl : [ 0.59  0.86  0.1   0.42]
A0034 :      Alzmr : [ 0.35  0.96  0.41  0.93]
A0768 :      Alzmr : [ 0.65  0.37  0.7   0.24]
A0240 :      Alzmr : [ 0.87  0.78  0.1   0.28]
A0042 :      Alzmr : [ 0.12  0.3   0.35  0.7 ]
A0141 :      Alzmr : [ 0.85  0.28  0.06  0.74]
A0888 :      Alzmr : [ 0.85  0.78  0.93  0.7 ]
A0032 :      Alzmr : [ 0.28  0.41  0.61  0.09]
A0596 :      Alzmr : [ 0.28  0.15  0.88  0.23]
A0969 :      Alzmr : [ 0.47  0.37  0.52  0.58]
A0215 :      Alzmr : [ 0.49  0.7   0.31  0.96]
A0074 :      Alzmr : [ 0.87  0.7   0.37  0.7 ]
A0229 :      Alzmr : [ 0.96  0.34  0.59  0.96]
M0760 :        MCI : [ 0.27  0.22  0.37  0.14]
M0434 :        MCI : [ 0.26  0.04  0.49  0.92]
M0033 :        MCI : [ 0.14  0.39  0.71  0.5 ]
```

```
M0942 :         MCI : [ 0.19  0.29  0.42  0.46]
M0034 :         MCI : [ 0.36  0.54  0.67  0.71]
M0868 :         MCI : [ 0.29  0.46  0.47  0.83]
M0595 :         MCI : [ 0.62  0.07  0.66  0.75]
M0476 :         MCI : [ 0.73  0.97  0.59  0.24]
M0770 :         MCI : [ 0.81  0.78  0.28  0.61]
M0577 :         MCI : [ 0.84  0.86  0.94  0.5 ]
M0638 :         MCI : [ 0.61  0.64  0.94  0.94]
M0421 :         MCI : [ 0.73  0.16  0.97  0.69]
M0006 :         MCI : [ 0.76  0.62  0.49  0.03]
M0552 :         MCI : [ 0.26  0.85  0.13  0.31]
M0040 :         MCI : [ 0.27  0.52  0.61  0.49]
M0165 :         MCI : [ 0.03  0.79  0.92  0.79]
M0256 :         MCI : [ 0.06  0.06  0.69  0.97]
M0127 :         MCI : [ 0.42  0.11  0.93  0.5 ]
```

Did you see that? *It's so intuitive and natural!* Such a clean traversal of dataset.

Thanks to the choice of the OrderedDict() to represent the data, classes and labels underneath, the order of sample addition is retained. Hence the correspondence across samples in the dataset not only key-wise (by the sample id), but also index-wise.

CHAPTER 8

Subject-wise tranform

Quite often, we are interested in computing some statistics on data for a given subject (such as mean, or ROI-wise median). Typically this requires a loop, with some computation and organizing it in a new dataset! A simple routine pattern of usage, but can't avoided if you are still fiddling with representing your dataset in medieval matrices! :).

If you organized your dataset in a `pyradigm`, such computation is trivial, thanks to builtin implementation of `transform` method. The mean value for each subject can be computed and organized in a new dataset, with an intuitive and single line:

```
mean_data = dataset.transform(np.mean)
mean_data.description = 'mean values per subject'
mean_data
```

```
mean values per subject
45 samples, 3 classes, 1 features.
Class Cntrl : 15 samples.
Class Alzmr : 12 samples.
Class   MCI : 18 samples.
```

As the transform accepts an arbitrary callable, we could do many more sophisticated things, such as access the subset of features e.g. cortical thickness for a particular region of interest (say posterior cingulate gyrus).

```
# let's make a toy function to return the indices for the ROI
def get_ROI_indices(x): return x[:3]
```

Using this "mask" function, we can easily obtain features for an ROI

```
pcg = dataset.transform(get_ROI_indices)
```

We can verify that the new dataset does indeed have only 3 features, for the same subjects/classes:

```
pcg
```

```
None
ADNI1 baseline: cortical thickness features from Freesurfer v4.3, QCed.
```

```
45 samples, 3 classes, 3 features.
Class Cntrl : 15 samples.
Class Alzmr : 12 samples.
Class   MCI : 18 samples.
```

```
pcg.num_features
```

```
3
```

Let's make a bar plot with the just computed numbers:

```
data, lbl, keys = pcg.data_and_labels()
```

```
n, bins, patches = plt.hist(data)
```

Remember as the original source of data was random, this has no units, property or meaning!

CHAPTER 9

Subset selection

In addition to the structured way of obtaining the various properties of this dataset, this implementation really will come in handy when you have to slice and dice the dataset (with large number of classes and features) into smaller subsets (e.g. for binary classification). Let's see how we can retrieve the data for a single class:

```
ctrl = dataset.get_class('Cntrl')
```

That's it, obtaining the data for a given class is a simple call away.

Now let's see what it looks like:

```
ctrl
```

```
 Subset derived from: ADNI1 baseline: cortical thickness features from Freesurfer v4.
↪3, QCed.
15 samples, 1 classes, 4 features.
Class Cntrl : 15 samples.
```

Even with updated description automatically, to indicate its history. Let's see some data from controls:

```
ctrl.glance(2)
```

```
{'C0408': array([ 0.49,  0.38,  0.05,  0.82]),
 'C0562': array([ 0.64,  0.59,  0.01,  0.8 ])}
```

We can also query a random subset of samples for manual inspection or cross-validation purposes. For example:

```
random_subset = dataset.random_subset(perc_in_class=0.3)
random_subset
```

```
 Subset derived from: ADNI1 baseline: cortical thickness features from Freesurfer v4.
↪3, QCed.
12 samples, 3 classes, 4 features.
Class Cntrl : 4 samples.
```

```
Class Alzmr : 3 samples.
Class   MCI : 5 samples.
```

You can see which samples were selected:

```
random_subset.sample_ids
```

```
['C0562',
 'C0565',
 'C0372',
 'C0492',
 'A0240',
 'A0032',
 'A0229',
 'M0034',
 'M0770',
 'M0552',
 'M0165',
 'M0127']
```

You can verify that it is indeed random by issuing another call:

```
# supplying a new seed everytime to ensure randomization
from datetime import datetime
dataset.random_subset(perc_in_class=0.3).sample_ids
```

```
['C0562',
 'C0822',
 'C0949',
 'C0141',
 'A0034',
 'A0141',
 'A0032',
 'M0434',
 'M0942',
 'M0868',
 'M0421',
 'M0552']
```

Let's see how we can retrieve specific samples by their IDs (for which there are many use cases):

```
data = dataset.get_subset(dataset.sample_ids[1:20])
data
```

```
 Subset derived from: ADNI1 baseline: cortical thickness features from Freesurfer v4.
↪3, QCed.
19 samples, 2 classes, 4 features.
Class Cntrl : 14 samples.
Class Alzmr : 5 samples.
```

So as simple as that.

9.1 Cross-validation

If you would like to develop a variant of cross-validation, and need to obtain a random split of the dataset to obtain training and test sets, it is as simple as:

```
train_set, test_set = dataset.train_test_split_ids( train_perc = 0.5)
```

This method returns two sets of sample ids corresponding to training set (which 50% of samples from all classes in the dataset) and the rest in test_set. Let's see what they have:

```
train_set, test_set
```

```
(['C0760',
  'C0822',
  'C0565',
  'C0170',
  'C0562',
  'C0141',
  'C0041',
  'A0768',
  'A0888',
  'A0032',
  'A0969',
  'A0141',
  'A0034',
  'M0434',
  'M0421',
  'M0577',
  'M0256',
  'M0127',
  'M0033',
  'M0760',
  'M0476',
  'M0165'],
 ['M0040',
  'A0240',
  'C0241',
  'C0492',
  'A0074',
  'A0042',
  'M0942',
  'M0595',
  'M0006',
  'C0372',
  'C0064',
  'C0557',
  'M0552',
  'M0034',
  'C0408',
  'C0980',
  'A0229',
  'C0949',
  'A0596',
  'M0770',
  'A0215',
  'M0868',
  'M0638'])
```

We can also get a train/test split by specifying an exact number of subjects we would like from each class (e.g. when you would like to avoid class imbalance in the training set):

```
train_set, test_set = dataset.train_test_split_ids( count_per_class = 3)
```

Let's see what the training set contains - we expect 3*3 =9 subjects :

```
train_set
```

```
['C0557',
 'C0041',
 'C0949',
 'A0768',
 'A0888',
 'A0229',
 'M0165',
 'M0476',
 'M0040']
```

We can indeed verify that is the case, by creating a new smaller dataset from that list of ids and getting a summary:

```
training_dataset = dataset.get_subset(train_set)
training_dataset
```

```
 Subset derived from: ADNI1 baseline: cortical thickness features from Freesurfer v4.
↪3, QCed.
9 samples, 3 classes, 4 features.
Class Cntrl : 3 samples.
Class Alzmr : 3 samples.
Class   MCI : 3 samples.
```

Another programmatic way to look into different classes is this:

```
class_set, label_set, class_sizes = training_dataset.summarize_classes()
class_set, label_set, class_sizes
```

```
(['MCI', 'Cntrl', 'Alzmr'], [2, 0, 1], array([ 3.,  3.,  3.]))
```

which returns all the classes that you could iterative over.

Using these two lists, we can easily obtain subset datasets, as illustrated below.

```
dataset
```

```
ADNI1 baseline: cortical thickness features from Freesurfer v4.3, QCed.
45 samples, 3 classes, 4 features.
Class Cntrl : 15 samples.
Class Alzmr : 12 samples.
Class   MCI : 18 samples.
```

```
binary_dataset = dataset.get_class(['Cntrl','Alzmr'])
binary_dataset
```

```
 Subset derived from: ADNI1 baseline: cortical thickness features from Freesurfer v4.
↪3, QCed.
27 samples, 2 classes, 4 features.
```

```
Class Cntrl : 15 samples.
Class Alzmr : 12 samples.
```

How about selecting a subset of features from all samples?

```
binary_dataset.get_feature_subset(range(2))
```

```
Subset features derived from:

 Subset derived from: ADNI1 baseline: cortical thickness features from Freesurfer v4.
↪3, QCed.
27 samples, 2 classes, 2 features.
Class Cntrl : 15 samples.
Class Alzmr : 12 samples.
```

Great. Isn't it? You can also see the two-time-point history (initial subset in classes, followed by a subset in features).

CHAPTER 10

Serialization

Once you have this dataset, you can save and load these trivially using your favourite serialization module. Let's do some pickling:

```
out_file = os.path.join(work_dir,'binary_dataset_Ctrl_Alzr_Freesurfer_thickness_v4p3.
↪MLDataset.pkl')
binary_dataset.save(out_file)
```

That's it - it is saved.

Let's reload it from disk and make sure we can indeed retrieve it:

```
reloaded = MLDataset(filepath=out_file) # another form of the constructor!
```

```
reloaded
```

```
 Subset derived from: ADNI1 baseline: cortical thickness features from Freesurfer v4.
↪3, QCed.
27 samples, 2 classes, 4 features.
Class Cntrl : 15 samples.
Class Alzmr : 12 samples.
```

We can check to see they are indeed one and the same:

```
binary_dataset == reloaded
```

```
True
```

CHAPTER 11

Dataset Arithmetic

You might wonder how can you combine two different types of features (thickness and shape) from the dataset. Piece of cake, see below ...

To concatenat two datasets, first we make a second dataset:

```
dataset_two = MLDataset(in_dataset=dataset) # yet another constructor: in its copy␣
↪form!
```

How can you check if they are "functionally identical"? As in same keys, same data and classes for each key... Easy:

```
dataset_two == dataset
```

```
True
```

Now let's try the arithmentic:

```
combined = dataset + dataset_two
```

```
Identical keys found. Trying to horizontally concatenate features for each sample.
```

Great. The add method recognized the identical set of keys and performed a horiz cat, as can be noticed by the twice the number of features in the combined dataset:

```
combined
```

```
45 samples, 3 classes, 8 features.
Class Cntrl : 15 samples.
Class Alzmr : 12 samples.
Class   MCI : 18 samples.
```

We can also do some removal in similar fashion:

```
smaller = combined - dataset
```

```
C0562 removed.
C0408 removed.
C0760 removed.
C0170 removed.
C0241 removed.
C0980 removed.
C0822 removed.
C0565 removed.
C0949 removed.
C0041 removed.
C0372 removed.
C0141 removed.
C0492 removed.
C0064 removed.
C0557 removed.
A0034 removed.
A0768 removed.
A0240 removed.
A0042 removed.
A0141 removed.
A0888 removed.
A0032 removed.
A0596 removed.
A0969 removed.
A0215 removed.
A0074 removed.
A0229 removed.
M0760 removed.
M0434 removed.
M0033 removed.
M0942 removed.
M0034 removed.
M0868 removed.
M0595 removed.
M0476 removed.
M0770 removed.
M0577 removed.
M0638 removed.
M0421 removed.
M0006 removed.
M0552 removed.
M0040 removed.
M0165 removed.
M0256 removed.
M0127 removed.
```

```
/Users/Reddy/dev/pyradigm/pyradigm/pyradigm.py:1169: UserWarning: Requested removal␣
↪of all the samples - output dataset would be empty.
  warnings.warn('Requested removal of all the samples - output dataset would be empty.
↪')
```

Data structure is even producing a warning to let you know the resulting output would be empty! We can verify that:

```
bool(smaller)
```

```
False
```

CHAPTER 12

Portability

This is all well and good. How does it interact with other packages out there, you might ask? It is as simple as you can imagine:

```
from sklearn import svm
clf = svm.SVC(gamma=0.001, C=100.)
```

```
data_matrix, target, sample_ids = binary_dataset.data_and_labels()
clf.fit(data_matrix, target)
```

```
SVC(C=100.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.001, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
```

There you have it, a simple example to show you the utility and convenience of this dataset.

Thanks for checking it out.

I would appreciate if you could give me feedback on improving or sharpening it further.

CHAPTER 13

API Reference

A tutorial-like presentation is available at *Usage examples*, using the following API.

class pyradigm.**MLDataset**(*filepath=None*, *in_dataset=None*, *arff_path=None*, *data=None*, *labels=None*, *classes=None*, *description=''*, *feature_names=None*, *encode_nonnumeric=False*)

Bases: object

An ML dataset to ease workflow and maintain integrity.

add_classes(*classes*)

Helper to rename the classes, if provided by a dict keyed in by the orignal keys

classes [dict] Dict of class named keyed in by sample IDs.

TypeError If classes is not a dict.

ValueError If all samples in dataset are not present in input dict, or one of they samples in input is not recognized.

add_sample(*sample_id*, *features*, *label*, *class_id=None*, *overwrite=False*, *feature_names=None*)

Adds a new sample to the dataset with its features, label and class ID.

This is the preferred way to construct the dataset.

sample_id [str, int] The identifier that uniquely identifies this sample.

features [list, ndarray] The features for this sample

label [int, str] The label for this sample

class_id [int, str] The class for this sample. If not provided, label converted to a string becomes its ID.

overwrite [bool] If True, allows the overwite of features for an existing subject ID. Default : False.

feature_names [list] The names for each feature. Assumed to be in the same order as *features*

ValueError If *sample_id* is already in the MLDataset (and overwrite=False), or If dimensionality of the current sample does not match the current, or If *feature_names* do not match existing names

TypeError If sample to be added is of different data type compared to existing samples.

classmethod check_features(*features*)
Method to ensure data to be added is not empty and vectorized.

features [iterable] Any data that can be converted to a numpy array.

features [numpy array] Flattened non-empty numpy array.

ValueError If input data is empty.

class_set
Set of unique classes in the dataset.

class_sizes
Returns the sizes of different objects in a Counter object.

classes
Identifiers (sample IDs, or sample names etc) forming the basis of dict-type MLDataset.

data
data in its original dict form.

data_and_labels()
Dataset features and labels in a matrix form for learning.

Also returns sample_ids in the same order.

data_matrix [ndarray] 2D array of shape [num_samples, num_features] with features corresponding row-wise to sample_ids

labels [ndarray] Array of numeric labels for each sample corresponding row-wise to sample_ids

sample_ids [list] List of sample ids

del_sample(*sample_id*)
Method to remove a sample from the dataset.

sample_id [str] sample id to be removed.

UserWarning If sample id to delete was not found in the dataset.

description
Text description (header) that can be set by user.

dtype
number of features in each sample.

extend(*other*)
Method to extend the dataset vertically (add samples from anotehr dataset).

other [MLDataset] second dataset to be combined with the current (different samples, but same dimensionality)

TypeError if input is not an MLDataset.

feature_names
Returns the feature names as an numpy array of strings.

get(*item*, *not_found_value=None*)
Method like dict.get() which can return specified value if key not found

get_class(*class_id*)
Returns a smaller dataset belonging to the requested classes.

class_id [str] identifier of the class to be returned.

MLDataset With subset of samples belonging to the given class.

ValueError If one or more of the requested classes do not exist in this dataset. If the specified id is empty or None

get_feature_subset(*subset_idx*)
Returns the subset of features indexed numerically.

subset_idx [list, ndarray] List of indices to features to be returned

MLDataset [MLDataset] with subset of features requested.

UnboundLocalError If input indices are out of bounds for the dataset.

get_subset(*subset_ids*)
Returns a smaller dataset identified by their keys/sample IDs.

subset_ids [list] List od sample IDs to extracted from the dataset.

sub-dataset [MLDataset] sub-dataset containing only requested sample IDs.

glance(*nitems=5*)
Quick and partial glance of the data matrix.

nitems [int] Number of items to glance from the dataset. Default : 5

dict

keys
Sample identifiers (strings) forming the basis of MLDataset (same as sample_ids)

static keys_with_value(*dictionary*, *value*)
Returns a subset of keys from the dict with the value supplied.

label_set
Set of labels in the dataset corresponding to class_set.

labels
Returns the array of labels for all the samples.

num_classes
Total number of classes in the dataset.

num_features
number of features in each sample.

num_samples
number of samples in the entire dataset.

random_subset(*perc_in_class=0.5*)
Returns a random sub-dataset (of specified size by percentage) within each class.

perc_in_class [float] Fraction of samples to be taken from each class.

subdataset [MLDataset] random sub-dataset of specified size.

random_subset_ids(*perc_per_class=0.5*)
Returns a random subset of sample ids (of specified size by percentage) within each class.

perc_per_class [float] Fraction of samples per class

subset [list] Combined list of sample ids from all classes.

ValueError If no subjects from one or more classes were selected.

UserWarning If an empty or full dataset is requested.

random_subset_ids_by_count(*count_per_class=1*)

Returns a random subset of sample ids of specified size by count, within each class.

count_per_class [int] Exact number of samples per each class.

subset [list] Combined list of sample ids from all classes.

sample_ids
Sample identifiers (strings) forming the basis of MLDataset (same as keys).

sample_ids_in_class(*class_id*)
Returns a list of sample ids belonging to a given class.

class_id [str] class id to query.

subset_ids [list] List of sample ids belonging to a given class.

save(*file_path*)
Method to save the dataset to disk.

file_path [str] File path to save the current dataset to

IOError If saving to disk is not successful.

summarize_classes()
Summary of classes: names, numeric labels and sizes

tuple : class_set, label_set, class_sizes

class_set [list] List of names of all the classes

label_set [list] Label for each class in class_set

class_sizes [list] Size of each class (number of samples)

train_test_split_ids(*train_perc=None*, *count_per_class=None*)
Returns two disjoint sets of sample ids for use in cross-validation.

Offers two ways to specify the sizes: fraction or count. Only one access method can be used at a time.

train_perc [float] fraction of samples from each class to build the training subset.

count_per_class [int] exact count of samples from each class to build the training subset.

train_set [list] List of ids in the training set.

test_set [list] List of ids in the test set.

ValueError If the fraction is outside open interval (0, 1), or If counts are outside larger than the smallest class, or If unrecongized format is provided for input args, or If the selection results in empty subsets for either train or test sets.

transform(*func*, *func_description=None*)

Applies a given a function to the features of each subject and returns a new dataset with other info unchanged.

func [callable] A valid callable that takes in a single ndarray and returns a single ndarray. Ensure the transformed dimensionality must be the same for all subjects.

If your function requires more than one argument, use *functools.partial* to freeze all the arguments except the features for the subject.

func_description [str, optional] Human readable description of the given function.

xfm_ds [MLDataset] with features obtained from subject-wise transform

TypeError If given func is not a callable

ValueError If transformation of any of the subjects features raises an exception.

Simple:

```
from pyradigm import MLDataset

thickness = MLDataset(in_path='ADNI_thickness.csv')
pcg_thickness = thickness.apply_xfm(func=get_pcg, description = 'applying ROI␣
↪mask for PCG')
pcg_median = pcg_thickness.apply_xfm(func=np.median, description='median per␣
↪subject')
```

Complex example with function taking more than one argument:

```
from pyradigm import MLDataset
from functools import partial
import hiwenet

thickness = MLDataset(in_path='ADNI_thickness.csv')
roi_membership = read_roi_membership()
hw = partial(hiwenet, groups = roi_membership)

thickness_hiwenet = thickness.transform(func=hw, description = 'histogram␣
↪weighted networks')
median_thk_hiwenet = thickness_hiwenet.transform(func=np.median, description=
↪'median per subject')
```

pyradigm.**cli_run**()

Command line interface

This is the command line interface

- to display basic info about datasets without having to code
- to perform basic arithmetic (add multiple classes or feature sets)

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index