Maciej Swat

PYTHON-BASED RAM REPORTING AND ANALYSIS FRAMEWORK

TUTORIAL AND DOCUMENTATION

Overview of the architecture of the framework

This technical document describes basic concepts behind architecture of the reporting and analysis framework user by RAM DARPA project. The task at hand can be summarized as follows: for each patient analyze iEEG data and produce summary report. The core of data analysis can be broken in several distinct steps:

1. Read in Behavioral data (so called events file)
2. Read in raw intracranial EEG data (iEEG)
3. Perform spectral decomposition of the iEEG data
4. Carry out additional tasks such as power computation, classifier fitting, plot generation report generation etc…

Instead of writing a sequential code that performs the above we introduce two Python classes: `RamPipeline` and `RamTask`.

`RamTask` is a base class that is used to implement single task, for example a task to read event files or a task to generate plots for the just analyzed data.

`RamPipeline`, is an object that could be thought of as a collection of tasks. Users add tasks to the `RamPipeline` and after all the tasks are in the pipeline. Users call `execute_pipeline` function and all the tasks are executed in the orders in which they were added. The pipeline takes care of outputting all results into workspace directory and greatly reduces the need to file path name wrangling. In addition, after execution of each task the pipeline marks this task as complete so that if user decides to run the pipeline again, the pipeline will skip the task. In some cases the task should be run each time the pipeline gets executed so users have the ability to mark tasks as such i.e. those tasks will not be marked as completed and the pipeline will always run them each time it gets executed.

Another concept that is useful is the idea of restoring the task. For example, imagine that a given task does some heavy computation, and the results of this computation are stored in a file. We do not want to call computational step each time we run the task and therefore we mark this task as complete after it finishes. But next time we execute the pipeline we would like to use the results that were stored on a disk during previous pipeline execution. To accomplish this all we need to do is to implement `restore` function that is a part of `RamTask` base class. When the pipeline is executed and the pipeline notices that a given task is complete it will try to call the `restore` function to give users an opportunity to load previously stored results.

Let's take a look at an example of the code to make things more concrete:

```
import numpy as np
from RamPipeline import RamPipeline

from EventPreparation import EventPreparation
from EEGRawPreparation import EEGRawPreparation
```

```
class ComputePipeline(RamPipeline):
    def __init__(self, workspace_dir):
        RamPipeline.__init__(self)
        self.set_workspace_dir(workspace_dir)


# sets up processing pipeline
compute_pipeline = ComputePipeline(workspace_dir='~/scratch/documentation_demo')

compute_pipeline.add_task(EventPreparation(mark_as_completed=False))

compute_pipeline.add_task(EEGRawPreparation(mark_as_completed=False))


# starts processing pipeline
compute_pipeline.execute_pipeline()
```

This is a fairly common use-case for the implementation of the pipeline object. We inherit from `RamPipeline` object and call `self.set_workspace_dir()` function that is a part of the `RamPipeline` base class. This function informs pipeline object to set up a workspace directory once the pipeline starts running. The idea here is that we want all of the output be stored in workspace-directory (and its subdirectories). We will also specify path for all outputs originating from within the pipeline object relative to the `workspace_dir`. This way a task that would like to create a file inside `workspace_dir` would use the following code:

```
self.create_file_in_workspace_dir(file_name='computation_folder/powers.npy', mode='w')
```

As a result of this call a file `~/scratch/documentation_demo /computation_folder/powers.npy` woud get created.

As you can see instead of specifying the full path you simply specify path relative to the workspace directory. When you work with many files this will save you a lot of effort to get path names right and your code will look much cleaner. We will talk about the functionality of the `RamTask` objects in a moment but first let's take a at the structure of the main Python script that creates pipeline object, adds task and executes the pipeline:

```
import numpy as np
from RamPipeline import RamPipeline

from EventPreparation import EventPreparation
from EEGRawPreparation import EEGRawPreparation


class ComputePipeline(RamPipeline):
    def __init__(self, workspace_dir):
        RamPipeline.__init__(self)
        self.set_workspace_dir(workspace_dir)


# sets up processing pipeline
compute_pipeline = ComputePipeline(workspace_dir='~/scratch/documentation_demo')

compute_pipeline.add_task(EventPreparation(mark_as_completed=False))

compute_pipeline.add_task(EEGRawPreparation(mark_as_completed=False))
```

```
# starts processing pipeline
compute_pipeline.execute_pipeline()
```

As one can see the basic idea is to create pipeline, create tasks, add tasks to pipeline and execute pipeline. The code looks quite simple. The number of arguments passed to task constructors is small, and tasks appear to be fully independent e.g. by looking at this code fragment you would probably not be aware that `EventPreparation` task communicates with `EEGRawPreparation` behind the scene. This is intentional. The top-level view should only expose bird-eye view of the processing pipeline, without presenting overwhelming amount of details.

Let us take a look at the `RamTask` object. When you open `RamTask.py` file the methods are quite well documented but it is instructional to show you they core structure of the `RamTask` class:

```
class RamTask(object):

    def __init__(self, mark_as_completed=True)

    def set_name(self,name):
    def name(self):
    def pass_object(self, name, obj):
    def get_passed_object(self, name):
    def get_task_completed_file_name(self):
    def is_completed(self):
    def set_pipeline(self, pipeline):
    def set_workspace_dir(self,workspace_dir):
    def set_mark_as_completed(self, flag):
    def open_file_in_workspace_dir(self,file_name, mode='r'):
    def create_file_in_workspace_dir(self, file_name, mode='w'):
    def create_multiple_files_in_workspace_dir(self, *rel_file_names, **options):
    def create_dir_in_workspace(self, dir_name):
    def create_multiple_dirs_in_workspace(self, *dir_names):
    def set_file_resources_to_copy(self, *file_resources, **kwds):
    def set_file_resources_to_move(self, *file_resources, **kwds):
    def make_dir_tree(self, dirname):
    def copy_file_resources_to_workspace(self):
    def move_file_resources_to_workspace(self):
    def get_path_to_resource_in_workspace(self, *rel_path_components):
    def get_pipeline(self):
    def get_workspace_dir(self):
    def set_pipeline(self,pipeline):
    def restore(self):
    def run(self):
```

Along with `RamPipeline` this class is the essence of the analysis and reporting framework. All tasks classes inherit from `RamTask` class. `RamTask` has multiple methods that facilitate file/directory operations, methods that facilitate passing objects from one task to another (`pass_object`, `get_passed_object`) and two essential methods that users reimplement in the derived class: run and restore.
Run is the method where the task's code gets implemented while restore (whose implementation is optional) allows users to e.g. skip computation step and read the results that the task saved during its previous run.

Each time the task's method run is successfully completed the pipeline object will create a file in the workspace directory that will have the following format" `task_name.completed`. For example if we had a

task with class name `ComputePowers` the pipeline would create `ComputePowers.completed` file in the workspace directory after `ComputePowers` was successfully completed.

It is worth saying here that we would like to have some tasks being run each time pipeline gets executed. To allow this all you need to is to pass `mark_as_completed=False` as an argument of the task's constructor. For example because `EventPreparation` task is constructed as :

```
EventPreparation(mark_as_completed=False)
```

it will be run each tie the pipeline gets executed. In contrast, if we create `ComputePowers` using:

```
ComputePowers(params=params, mark_as_completed=True)
```

it will be run only once. Each subsequent executions of the pipeline will skip `ComputePowers`' run function. This bring us the `restore` method.

Restoring state of the task

As you probably suspect the restore method is use to take advantage of the task results that were computed during first execution of the pipeline and stored on a disk. For example `ComputePowers` runs pretty extensive computations first time it is executed and saves results. All subsequent runs will read those results instead of spending time in the lengthy computations. Let's look at the concrete example:

```
class ComputePowers(RamTask):
    def __init__(self, params, mark_as_completed=True):
        RamTask.__init__(self, mark_as_completed)
        self.params = params
        self.pow_mat = None

    def restore(self):
        # loading previously computed power array
        self.pow_mat = joblib.load(self.get_path_to_resource_in_workspace
                       (pow_mat.pkl'))

        #passing object and making it available to subsequent tasks
        self.pass_object('pow_mat', self.pow_mat)

    def run(self):

        #fetching event array that was passed from earlier task
        events = self.get_passed_object(task+'_events')

        # this call will take significant time to complete
        self.pow_mat = self.compute_powers(events)

        #writing power array to the disk
        joblib.dump(self.pow_mat, self.get_path_to_resource_in_workspace
        ('pow_mat.pkl'))

        #passing object and making it available to subsequent tasks
        self.pass_object('pow_mat', self.pow_mat)

    def compute_powers(self,events):
        #this function takes a long time to finish
```

```
        ...
```

The `ComputePowers` class defines 3 methods: run, restore and compute powers. When the pipeline encounters `ComputePowers` task for the first time it will notice that there is no marker for completed `ComputePowers` task in the workspace directory. In this case run function will be called. First thing we do in the run function we fetch events record array that is passed from earlier tasks then we pass those events to the `compute_powers` functions where the actual computations are done. After `compute_powers` finishes it returns array spectral powers (`self.pow_mat`). We serialize (save to the hard drive) this array using dump function of the joblib module. You may equally well use numpy writer, HDF5 writer or NetCDF4 writer or use any of your favorite data output formats but for this example we chose to use joblib's format. At this point to make it available to subsequent tasks in the pipeline we use

```
self.pass_object('pow_mat', self.pow_mat).
```

Notice that we use `self.get_path_to_resource_in_workspace` function that is a member of the RamTask base class and it returns full path to the resource (file) located in the workspace directory. Again, we simple call instead of lengthy and messy expressions for path name wrangling.
Now imagine you are running this task second time i.e. the powers are computed and saved to the hard drive and the marker of the task completion (i.e. `ComputePowers.completed` file) is in the workspace directory. When pipeline executes a task that is marked as completed the pipeline will try restoring it. I.e. it will call restore function of the task. Recall that the implementation of the restore function was optional and for tasks that do not implement this function recall function is empty which means that in this case the pipeline will call empty function. On the other hand if the implementation of the recall function is non-empty i.e. if there is actual code in the restore function it will get executed when the pipeline executes task what was marked as completed. In the presented example the time to compute powers was approx. 2 hours on a single CPU whereas reading saved powers (roughly speaking 1 GB file took few seconds).

Reading in Event files

The so called event files are Matlab arrays of records that contain behavioral data about the experiment. Think of them as collection of data entries ordered by the timestamp that describe events that took place during experiment. The examples of events include presentation of the word, presentation of the math problem, brain stimulation etc… . When you open a sample event file (e.g. `/data/events/RAM_FR1/R1060M_events.mat`) in Matlab you will see the following screen:

Variables – events

events

1x2359 struct with 23 fields

Fields	subject	session	list	serialpos	type	item	itemno	recalled	mstime	msoffset	rectime	intrusion	isStim	expVersion	stimLo
23	'R1060M'	0	1	-999	'TRIAL'	'X'	-999	-999	1.4378e+...	0	-999	-999	-999	'v_1.05'	'X'
24	'R1060M'	0	1	-999	'COUNTD...	'X'	-999	-999	1.4378e+...	0	-999	-999	-999	'v_1.05'	'X'
25	'R1060M'	0	1	-999	'COUNTD...	'X'	-999	-999	1.4378e+...	0	-999	-999	-999	'v_1.05'	'X'
26	'R1060M'	0	1	-999	'ORIENT'	'X'	-999	-999	1.4378e+...	1	-999	-999	-999	'v_1.05'	'X'
27	'R1060M'	0	1	1	'WORD'	'DUCK'	82	1	1.4378e+...	1	3270	-999	0	'v_1.05'	'X'
28	'R1060M'	0	1	2	'WORD'	'MUG'	162	0	1.4378e+...	1	-999	-999	0	'v_1.05'	'X'
29	'R1060M'	0	1	3	'WORD'	'PIPE'	185	1	1.4378e+...	1	6288	-999	0	'v_1.05'	'X'
30	'R1060M'	0	1	4	'WORD'	'APE'	2	0	1.4378e+...	1	-999	-999	0	'v_1.05'	'X'
31	'R1060M'	0	1	5	'WORD'	'WORM'	298	0	1.4378e+...	1	-999	-999	0	'v_1.05'	'X'
32	'R1060M'	0	1	6	'WORD'	'PLATE'	189	0	1.4378e+...	1	-999	-999	0	'v_1.05'	'X'
33	'R1060M'	0	1	7	'WORD'	'WEB'	290	0	1.4378e+...	1	-999	-999	0	'v_1.05'	'X'
34	'R1060M'	0	1	8	'WORD'	'DEER'	72	0	1.4378e+...	1	-999	-999	0	'v_1.05'	'X'
35	'R1060M'	0	1	9	'WORD'	'BEAN'	16	1	1.4378e+...	1	23102	-999	0	'v_1.05'	'X'
36	'R1060M'	0	1	10	'WORD'	'MOTH'	158	0	1.4378e+...	1	-999	-999	0	'v_1.05'	'X'
37	'R1060M'	0	1	11	'WORD'	'SEED'	214	0	1.4378e+...	1	-999	-999	0	'v_1.05'	'X'
38	'R1060M'	0	1	12	'WORD'	'MAT'	152	0	1.4378e+...	1	-999	-999	0	'v_1.05'	'X'
39	'R1060M'	0	1	-999	'DISTRAC...	'X'	-999	-999	1.4378e+...	0	-999	-999	-999	'v_1.05'	'X'
40	'R1060M'	0	1	-999	'DISTRAC...	'X'	-999	-999	1.4378e+...	0	-999	-999	-999	'v_1.05'	'X'
41	'R1060M'	0	1	-999	'REC_STA...	'X'	-999	-999	1.4378e+...	-999	-999	-999	-999	'v_1.05'	'X'
42	'R1060M'	0	1	-999	'REC_WO...	'DUCK'	82	-999	1.4378e+...	-999	3270	0	-999	'v_1.05'	'X'
43	'R1060M'	0	1	-999	'REC_WO...	'MUD'	-1	-999	1.4378e+...	-999	4859	-1	-999	'v_1.05'	'X'
44	'R1060M'	0	1	-999	'REC_WO...	'PIPE'	185	-999	1.4378e+...	-999	6288	0	-999	'v_1.05'	'X'
45	'R1060M'	0	1	-999	'REC_WO...	'BEAN'	16	-999	1.4378e+...	-999	23102	0	-999	'v_1.05'	'X'
46	'R1060M'	0	1	-999	'REC_END'	'X'	-999	-999	1.4378e+...	1	-999	-999	-999	'v_1.05'	'X'
47	'R1060M'	0	2	-999	'TRIAL'	'X'	-999	-999	1.4378e+...	0	-999	-999	-999	'v_1.05'	'X'
48	'R1060M'	0	2	-999	'COUNTD...	'X'	-999	-999	1.4378e+...	0	-999	-999	-999	'v_1.05'	'X'
49	'R1060M'	0	2	-999	'COUNTD...	'X'	-999	-999	1.4378e+...	0	-999	-999	-999	'v_1.05'	'X'
50	'R1060M'	0	2	-999	'ORIENT'	'X'	-999	-999	1.4378e+...	1	-999	-999	-999	'v_1.05'	'X'
51	'R1060M'	0	2	1	'WORD'	'MULE'	163	1	1.4378e+...	1	4722	-999	0	'v_1.05'	'X'

From that you can see that each event record consists of fields such as `subject`, `session`, `list`, `item`, `type`, `recalled`, etc… By filtering events we can pick events that e.g. correspond to session 0 , are of type "WORD" – which means they correspond to word stimulus being presented on the screen of the task laptop and belong to list number greater than 5. After you select those events you can read the raw eeg data corresponding to them and analyze it.

Let us focus on event reading task in Python. This is how we can write a task that reads events:

```
from RamPipeline import *
from EventReader import EventReader

class EventPreparation(RamTask):
    def __init__(self, mark_as_completed=True):
        RamTask.__init__(self, mark_as_completed)

    def run(self):

        e_reader =EventReader(
                  event_file='/Volumes/rhino_root/data/events/RAM_FR1/R1060M_events.mat',
                  eliminate_events_with_no_eeg=True,
                  data_dir_prefix='/Volumes/rhino_root'
                  )

        e_reader.read()

        events = e_reader.get_output()
        self.pass_object('FR1_events',events)
```

We first import `RamPipeline` module (this imports `RamPipeline` and `RamTask` classes) and `EventReader` class. We create a class `EventPreparation` that inherits from `RamTask`. In the `run` function we create an instance of `EventReader` class which is a Python object that knows how to read an event file stored in the Matlab format. `EventReader` takes as an argument path to event file (`event_file='/data/events/RAM_FR1/R1060M_events.mat'`), a flag whether to eliminate events with no eeg data associated with them (`eliminate_events_with_no_eeg=True`) and the data directory prefix. The data directory prefix is the name of the directory where you mount rhino directory. Th reason we specify it here is because when events are read the EventReader atach a special object to each event that knows how to read binary eeg data. This object needs to know location of the eeg files associated with each

event. The file to eeg data is stored in each event but unfortunately this is the absolute path on rhino cluster, thus when reading events on machines different than rhino we need to manipulate this path to point to the correct location. So if rhino file system is mounted under `/Volumes/rhino_root` the actual path to the eeg data stored in the read events should be
`/Volumes/rhino_root/<rhino_absolute_path_to_eegfile>`

Therefore if you are on OSX you may want to ssh-mount the whole rhino file system under `/Volumes/rhino_root`. Then you would set

```
data_dir_prefix='/Volumes/rhino_root'
```

After you create `EventReader` object (we called it `e_reader`) you call `read()` function from `e_reader` and it reads and parses Matlab event file. Once you call `e_reader.get_output()` you will get a Python object (of type `Events`) that has all the event information at your fingertips. Class `Events` is defined in the PTSA the PTSA package and if you look at its definition it is a thin wrapper on top of numpy `recarray` class, which for all our practical purposes means that we can treat it as a recarray. However, `Events` class has extra methods called `get_data` which we will later use to read raw eeg data. To summarize `Events` is a `recarray` that stores events and knows how to read raw eeg data.

Finally, the call to the `EventReader` function

```
self.pass_object('FR1_events',events)
```

makes events available for other tasks in the pipeline. Now that we know how to read events let explore few ways to display event information and then how to filter the events.

I will assume that all the print statements are being types right after the line where we assign events variable – see line with `# all the print statements go here` comment:

```
from RamPipeline import *
from EventReader import EventReader

class EventPreparation(RamTask):
    def __init__(self, mark_as_completed=True):
        RamTask.__init__(self, mark_as_completed)

    def run(self):

        e_reader = EventReader(event_file='/data/events/RAM_FR1/R1060M_events.mat',
                               eliminate_events_with_no_eeg=True,
                               data_dir_prefix='')

        e_reader.read()

        events = e_reader.get_output()

        # all the print statements go here
        print events


        self.pass_object('FR1_events',events)
```

In this case Python will output:

```
(u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018', 7926, array([], dtype=float64), -999,
-999, u'X', -999, -999, 0, 1437834209102.0, -999, -999, -999, 0, nan, nan, array([],
dtype=float64), nan, array([], dtype=float64), -999, u'X', u'R1060M', u'B',
<ptsa.data.rawbinwrapper.RawBinWrapper object at 0x10a711f10>)
 (u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018', 7991, u'v_1.05', -999, -999, u'X', -
999, -999, 0, 1437834209233.0, -999, -999, -999, 0, nan, nan, array([], dtype=float64),
nan, array([], dtype=float64), -999, u'X', u'R1060M', u'SESS_START',
<ptsa.data.rawbinwrapper.RawBinWrapper object at 0x10a711f50>)
 (u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018', 81260, u'v_1.05', -999, -999, u'X',
-999, -999, 0, 1437834355779.0, -999, -999, -999, 0, nan, nan, array([], dtype=float64),
nan, array([], dtype=float64), -999, u'X', u'R1060M', u'COUNTDOWN_START',
<ptsa.data.rawbinwrapper.RawBinWrapper object at 0x10a711f90>)
 ...,
 (u'/data10/RAM/subjects/R1060M/eeg.reref/R1060M_12Aug15_1425', 825816, u'v_1.05', -999,
-999, u'X', -999, 16, 0, 1439405932587.0, -999, -999, -999, 3, nan, nan, array([],
dtype=float64), nan, array([], dtype=float64), 0, u'X', u'R1060M', u'TRIAL',
<ptsa.data.rawbinwrapper.RawBinWrapper object at 0x10b512250>)
 (u'/data10/RAM/subjects/R1060M/eeg.reref/R1060M_12Aug15_1425', 825815, u'v_1.05', -999,
-999, u'X', -999, 16, 0, 1439405932586.0, -999, -999, -999, 3, nan, nan, array([],
dtype=float64), nan, array([], dtype=float64), -999, u'X', u'R1060M', u'COUNTDOWN_START',
<ptsa.data.rawbinwrapper.RawBinWrapper object at 0x10b512290>)
 (u'/data10/RAM/subjects/R1060M/eeg.reref/R1060M_12Aug15_1425', 8608, u'v_1.05', -999, -
999, u'X', -999, 16, 0, 1439404298079.0, -999, -999, -999, 3, nan, nan, array([],
dtype=float64), nan, array([], dtype=float64), -999, u'X', u'R1060M', u'E',
<ptsa.data.rawbinwrapper.RawBinWrapper object at 0x10b5122d0>)]
```

The reason the output is truncated is because the number of events in this sample is large and Pythn , similarly to Matlab truncates the output to the screen for the large arrays. To find out how many events we have in the events array just type

```
print events.shape
```

In the case of the R1060M patient for RAM_FR1 task Python will print:

```
(2357,)
```

This means that there are 2357 events.

Now, let us print the item field for each event:

```
print events['item']
```

we will get the following:

```
[u'X' u'X' u'X' ..., u'X' u'X' u'X']
```

Again, we have 2357 items so the output gets truncated. If we print item field for events 0-50 we will get full output:

```
[u'X' u'X' u'X' u'X' u'X' u'X' u'X' u'X' u'X' u'X' u'X' u'X' u'X' u'X' u'X'
 u'X' u'X' u'X' u'X' u'X' u'X' u'X' u'X' u'X' u'DUCK' u'MUG' u'PIPE' u'APE'
 u'WORM' u'PLATE' u'WEB' u'DEER' u'BEAN' u'MOTH' u'SEED' u'MAT' u'X' u'X'
```

```
 u'X' u'DUCK' u'MUD' u'PIPE' u'BEAN' u'X' u'X' u'X' u'X' u'X' u'MULE'
 u'HORSE']
```

Similarly printing eegfile using

```
print events['eegfile'][0:20]
```

will give us

```
[u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018'
 u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018'
 u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018'
 u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018'
 u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018'
 u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018'
 u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018'
 u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018'
 u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018'
 u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018'
 u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018'
 u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018'
 u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018'
 u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018'
 u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018'
 u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018'
 u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018'
 u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018'
 u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018'
 u'/data/eeg/R1060M/eeg.reref/R1060M_25Jul15_1018']
```

Now that we know how to print event information to the screen let's figure out how to filter events. Out first filtering criterion is to select only those events that correspond to word presentation on the test computer screen. In other words we are looking to select those rows of the events array that have type record field equal to "WORD":

```
print events[events['type']=='WORD'].shape
```

and the output is :

```
(1080,)
```

What is going on here? First we are constructing a Boolean array :

```
events['type']=='WORD'
```

and if we print it we will get:

```
[False False False False False False False False False False False False
 False False False False False False False False False False False False
  True  True  True  True  True  True  True  True  True  True  True  True
 False False False False False False False False False False False False
  True  True…]
```

When we substitute this Boolean array as an index of events it will select those rows of the events array that correspond to the `True` value of the Boolean array aka indicator array.

IMPORTANT: You cannot replace True False values in the indicator array with 1's and 0's If you do that you will end up with an array whose elements will consist of only 0^{th} and 1^{st} element of the events array. IF you have an array that has zeros and ones and you would like to use it as an indicator array you need to cast it to Boolean array using code similar to the one shown below:

```
a =np.array([0,1,0,0,1,1,0])
np.array(a,dtype=np.bool_)
```

Now let's take a look at a more complex selection rule. Suppose I want to select events that correspond to type of the event being `"WORD"` and the word should be recalled. Here is the code snippet that accomplishes this:

```
sel_events = events[(events.type=='WORD') & (events.recalled==1)]
```

In this case we construct our index array by executing logical `'and'` (`&` operator) between two index arrays :

```
(events.type=='WORD')
```

and

```
(events.recalled==1)
```

The result is selector array that picks required events.

Reading EEG Data

Most (but not all) records in the events array will have 'eegfile' entry that points to the location of the eeg files that contain raw eeg time series stored in the binary format.

To read those time series we will use PTSA Python package (https://github.com/compmem/ptsa) that knows how to read the raw eeg files.

Let us create a task that reads events:

```
from RamPipeline import *

class EEGRawPreparation(RamTask):
    def __init__(self, mark_as_completed=False):
        RamTask.__init__(self, mark_as_completed)

    def run(self):
        events = self.get_passed_object('FR1_events')

        sel_events = events[(events.type == 'WORD') & (events.recalled == 1)]

        sel_events = sel_events[0:20]
```

```
        eegs = sel_events.get_data(channels=['002','003'], start_time=-1.0, end_time=1.6,
                                   buffer_time=1.0, eoffset='eegoffset',
keep_buffer=True, eoffset_in_time=False)
```

Let us describe what is going on here. First we get events that are passed from the earlier task:

```
events = self.get_passed_object('FR1_events')
```

We assume that no selection has been performed on the events and the next line does the selection:

```
sel_events = events[(events.type == 'WORD') & (events.recalled == 1)]
```

Subsequently we select first 20 of the selected events i.e. recalled events that correspond to the presentation of a word on the task laptop.

Once we have events we can use capabilities of the `Events` class to read raw eeg data as implemented in the `get_data()` function. `get_data` function takes as arguments an array of channels , `start_time` i.e. the location of the first time series point in relation to the event timestamp. For example when we set

```
start_time=-1.0
```

this means that for each event in the `sel_events` array the read of the time series will start 1.0 second before the actual event time stamp. `end_time` describes the time offset of the last time-series data-point. Now, we use

```
buffer_time=1.0
```

This means that the data read will begin 1.0 second before `start_time` i.e. 2.0 seconds before event timestamp and will go 1.0 seconds after the `end_time` timestamp i.e. to 2.6 seconds after event timestamp. So for givens specification of times (start, end buffer) we will read 4.6 seconds of data. Now, the flag `keep_buffer` is used by reader to decide whether extra data as specified by the `buffer_time` settings will be kept or discrarded when the `get_data` function returns. Finally the argument

```
Eoffset = 'eegoffset'
```

tells eeg reader that the field in the `Events` recarray that stores information about location of the raw eegfile is called `'eegoffset'`.

The output `eegs` is of type `TimeSeries` which is a part of PTSA framework. Let us explore a bit. First thing we need to know is that TimeSeries is a 3D array where axes 0 is channels axis, axis 1 is events axis and axis 2 is time axis. Thus to get time series corresponding to channel '002' (axis 0, index 0), for the event number 10 (axis 1, index 11) we would write

```
print eegs[0][11]
```

and the result would be

```
[ 3830.72558   3845.080832  3856.777704 ...,  3926.42726   3925.895584
```

```
  3923.76888 ]
```

Making Plots

Very often when you compete analyses of data you would like to create some plots. Here you have many options and plotting packages. IN this example we will however show you how to create plots using relatively simple syntax. Internally we use matplotlib but instead of asking users to write lengthy plot descriptions of the plot we created convenience classes and functions that make plots ctreation much simpler. Surely, when you rely on our API you would sacrifice some flexibility but the code you write will be much shorter. Still, you will be able to tweak your plot however you like using maptplot lib functionality because our API returns matplotlib plot objects.

Let's see a simple example. Assume we are running the analysis and we already have a data (completed using previous tasks) that we want to plot. To generate plots as a part of the pipeline we create new task – `PlotTask`:

```
from RamPipeline import *
from PlotUtils import *
import numpy as np


class PlotTask(RamTask):
    def __init__(self, mark_as_completed=True):
        RamTask.__init__(self, mark_as_completed)

    def run(self):
        #prepare data sample
        x1 = np.arange(10)
        y1 = 2 * x1 ** 2 + 1
        yerr1 = np.random.rand(10)

        #create PlotData - a convenient way to gather all information about data series
        plot_data_1 = PlotData(x=x1, y=y1, yerr=yerr1, xlabel='attempt number',
                               ylabel='score')

        #create Panel Plot (panel contains only 1 plot)
        panel_plot = PanelPlot(i_max=1, j_max=1, title='Scores')

        # add plot data to the (0,0) location in the panel plot
        panel_plot.add_plot_data(0, 0, plot_data=plot_data_1)

        #generate plot - plot is a matplotlib.pyplot module
        plot = panel_plot.generate_plot()

        #save plot as pdf
        plot.savefig('scores.pdf', dpi=300, bboxinches='tight')
```

and for completeness the "pipeline Python file" looks as follows:

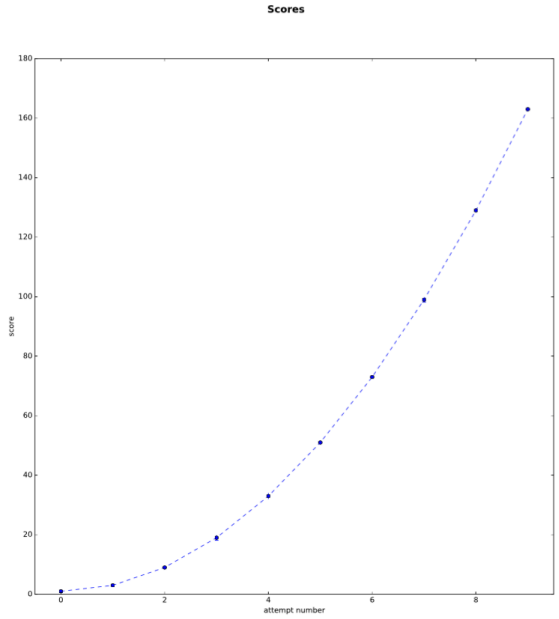
```
from RamPipeline import RamPipeline
from EventPreparation import EventPreparation
from EEGRawPreparation import EEGRawPreparation
from PlotTask import PlotTask
```

```
class ComputePipeline(RamPipeline):
    def __init__(self, workspace_dir):
        RamPipeline.__init__(self)
        self.set_workspace_dir(workspace_dir)

# sets up processing pipeline
compute_pipeline = ComputePipeline(workspace_dir='~/scratch/documentation_demo')

compute_pipeline.add_task(EventPreparation(mark_as_completed=False))

compute_pipeline.add_task(EEGRawPreparation(mark_as_completed=False))

compute_pipeline.add_task(PlotTask(mark_as_completed=False))

# starts processing pipeline
compute_pipeline.execute_pipeline()
```

As you can see in the run function of the PlotTask class we first create data series which consists of x, y and yerr numpy arrays. We then pass these arrays to the PlotUtils object called PlotData. The reason we do this will become obvious soon. In essence we want to aggregate all the information about data series and how to plot it in one single object. In our case PlotData instance does not include formatting information but we will show in the next example how to include those.

Important: Keep in mind that the presented plotting utilities were designed for the purpose of the reporting framework. They provide rudimentary ploting capabilities and if you really need highly sophisticated/customized plot your best bet is to use matplotlib directly.

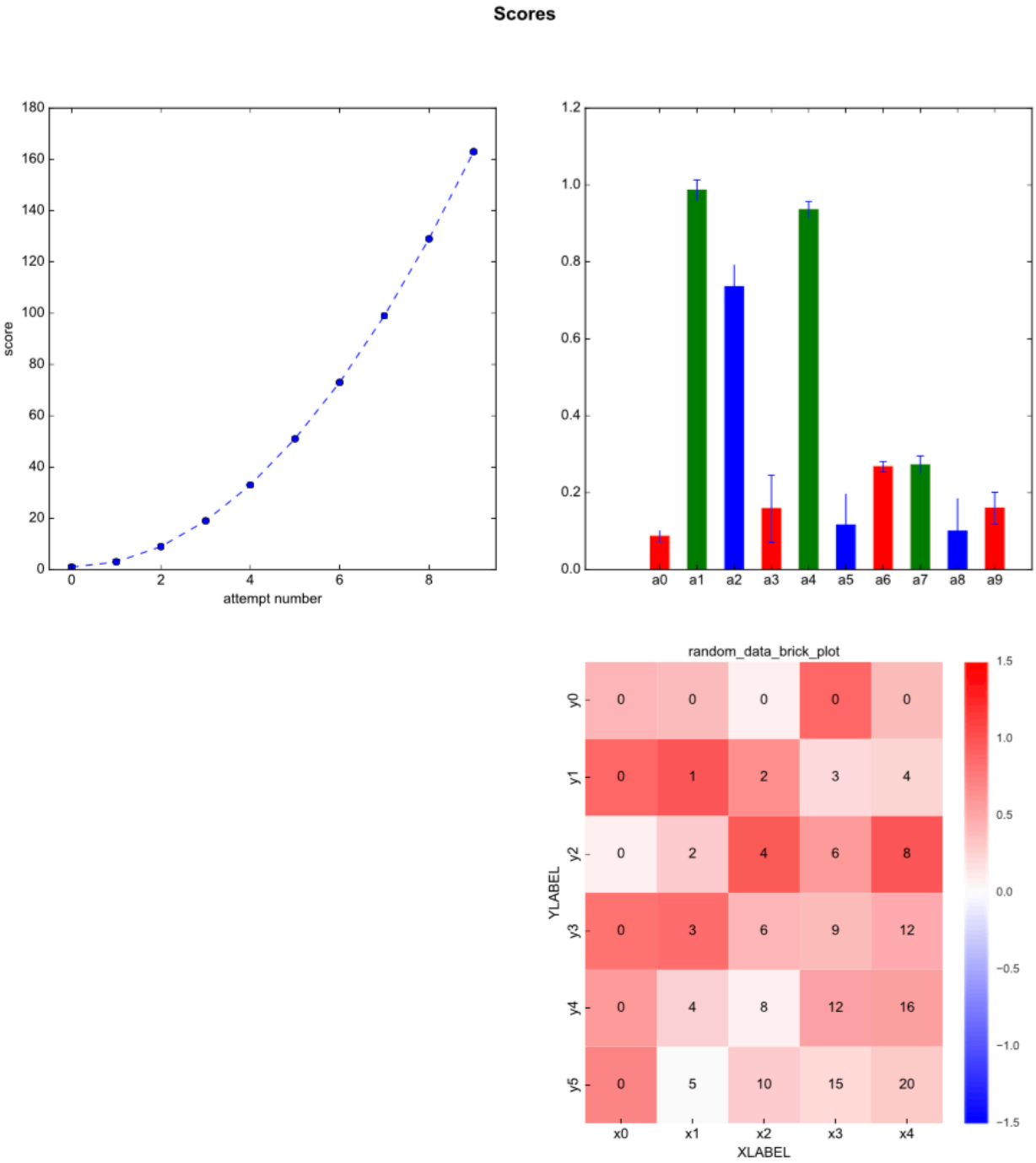
After creating PlotData instance we create PanelPlot instance. Panel plot instance is used to accommodate multiple plots in a single plot-sheet. Here our panel has only one plot but we will show how to make multiple plots in a single panel in the next example. Once we have PanelPlot object we can add PlotData instance to it (we need to specify panel coordinates – here they are (0,0)) and then we generate actual plot and as the last line in the run function we save the plot. The result is shown in figure below:



Let us look at a more complex example. To do this we will modify PlotTask a little (we are showing run function only):

```
    def run(self):
        # prepare data sample
        x1 = np.arange(10)
        y1 = 2 * x1 ** 2 + 1
        yerr1 = np.random.rand(10)

        # create PlotData - a convenient way to gather all information about data series
        plot_data_1 = PlotData(
        x=x1, y=y1, yerr=yerr1, xlabel='attempt number', ylabel='score',
        linestyle='dashed',
        color='r', marker='s'
        )

        # create plot data for Bar plotting
        plot_data_2 = BarPlotData(
        x=np.arange(10), y=np.random.rand(10),
        title='data01', yerr=np.random.rand(10) * 0.1,
        x_tick_labels=['a0','a1','a2','a3','a4','a5','a6','a7','a8','a9'],
        barcolors=['r','g','b','r','g','b','r','g','b','r']
        )

        data_frame = np.random.rand(6, 5)
        annotation_dictionary = {(0, 0): 10, (1, 2): 20}
        from itertools import product

        annotation_dictionary = \
        {(i, j): i * j for i, j in product(range(6), range(6))}

        x_tick_labels = ['x0', 'x1', 'x2', 'x3', 'x4']
        y_tick_labels = ['y0', 'y1', 'y2', 'y3', 'y4', 'y5']

        # create plot data for BrickHeatmap plotting
        plot_data_3 = BrickHeatmapPlotData(
        df=data_frame, annot_dict=annotation_dictionary,
        title='random_data_brick_plot',
        x_tick_labels=x_tick_labels, y_tick_labels=y_tick_labels,
        xlabel='XLABEL', ylabel='YLABEL',
        val_lim=[-1.5, 1.5]
        )


        # create Panel Plot (panel contains only 1 plot)
        panel_plot = PanelPlot(i_max=2, j_max=2, title='Scores')

        # add plot data to the (0,0) location in the panel plot
        panel_plot.add_plot_data(0, 0, plot_data=plot_data_1)

        # add bar plot data to the (0,1) location in the panel plot
        panel_plot.add_plot_data(0, 1, plot_data=plot_data_2)

        # add brick heratmap plot data to the (0,1) location in the panel plot
        panel_plot.add_plot_data(1, 1, plot_data=plot_data_3)

        # generate plot - plot is a matplotlib.pyplot module
        plot = panel_plot.generate_plot()

        # save plot as pdf
        plot.savefig('scores.pdf', dpi=300, bboxinches='tight')
```

When you run this script you will get the following plot:



Let's first understand placing of the plots in the panel. The panel coordinates follow "numpy convention" meaning that in tuple (i,j) `i` denotes row and `j` denotes column and (0,0) is in the upper left corner of the panel:

```
        # grid layout:
        ----------------------------------
        |
        |   (0,0)       (0,1)       (0,2)
        |
        |
        |
        |
        |   (1,0)       (1,1)       (1,2)
        |
        |
        |
        |
        |   (2,0)       (2,1)       (2,2)
        |
        ----------------------------------
```

To draw a bar plot we create BarPlotData:

```
        plot_data_2 = BarPlotData(
        x=np.arange(10), y=np.random.rand(10),
        title='data01', yerr=np.random.rand(10) * 0.1,
        x_tick_labels=['a0','a1','a2','a3','a4','a5','a6','a7','a8','a9'],
        barcolors=['r','g','b','r','g','b','r','g','b','r']
        )
```

where `x's` are consecutive integers `y's` are some values we want to graph, `yerr`, represents y error bars `x_tick_labels` represent the text that will be inserted under each bar and the dimension of the `x_tick_labels` should be the same as the dimension of the x data series. Finally, we can specify (`barcolors`) bar colors using matplotlib color codes (http://matplotlib.org/api/colors_api.html) and the size of this array should be the same as the `x` data series.

The third plot we are showing (notice, that panel location (1,0) is unoccupied) is a "brick" heatmap that might be useful to visualize 2D data series if we do not want to use 3D plotting.

```
        data_frame = np.random.rand(6, 5)
        annotation_dictionary = {(0, 0): 10, (1, 2): 20}
        from itertools import product

        annotation_dictionary = \
        {(i, j): i * j for i, j in product(range(6), range(6))}

        x_tick_labels = ['x0', 'x1', 'x2', 'x3', 'x4']
        y_tick_labels = ['y0', 'y1', 'y2', 'y3', 'y4', 'y5']

        # create plot data for BrickHeatmap plotting
        plot_data_3 = BrickHeatmapPlotData(
        df=data_frame, annot_dict=annotation_dictionary,
        title='random_data_brick_plot',
        x_tick_labels=x_tick_labels, y_tick_labels=y_tick_labels,
        xlabel='XLABEL', ylabel='YLABEL',
        val_lim=[-1.5, 1.5]
        )
```

We first create data series which is a 2D array put in the pandas dataframe

Next we create annotation dictionary of the following form:

```
(i,j):value
```

where `i, j` label rows and columns and values can be either numbers (as presented here) or a string. We use annotation dictionary to annotate individual bricks in the plot. `x_tick_labels` and `y_tick_labels` are use to label `x` and `y` axis of the brick array. `xlabel` and `ylabel` label titles of `x` and `y` axes respectively. `val_lim` is used to set scaling range for the colormap plotting.

The presented plotting capabilities are quite basic and for more complex plot types you should use matplotlib directly. Once we determined that certain plot type is used on a regular basis it will make sense to add it to our framework to simplify coding and standardize plot appearance.

Text Templating

On task that pops up quite frequently frequently in e.g. preparing series of reports is generating meta of markup files which are further converted to final documents. For example, based on our analysis we may want to generate series of TeX files (one per subject) where 95% of the content is "boiler plate" and 5% of the content is patient specific and may include information such as subject code, number of sessions, average inter stimulation interval etc…
Instead of "coding" the content off the entire latex document in Matlab , Python or C++ it is much better to prepare a standalone TeX file, and introduce a markup (placeholders) where replacements are needed and save it with .tex.tpl extension to indicate that this is a template file. For example our tex template may look like this:

```
\documentclass[a4paper]{article}
\usepackage[usenames,dvipsnames,svgnames,table]{xcolor}
\usepackage{graphicx,multirow}
\usepackage{epstopdf}
\usepackage{subfigure,amsmath}
\usepackage{wrapfig}
\usepackage{longtable}
\usepackage{pdfpages}
\usepackage{mathtools}
\usepackage{array}
\usepackage{enumitem}
\usepackage{booktabs}
\usepackage{sidecap} \usepackage{soul}
\usepackage[small,bf,it]{caption}
\setlength\belowcaptionskip{2pt}

\addtolength{\oddsidemargin}{-.875in}
\addtolength{\evensidemargin}{-.875in}
\addtolength{\textwidth}{1.75in}
\addtolength{\topmargin}{-.75in}
\addtolength{\textheight}{1.75in}

\newcolumntype{C}[1]{>{\centering\let\newline\\\arraybackslash\hspace{0pt}}m{#1}}
\usepackage{fancyhdr}
\pagestyle{fancy}
\fancyhf{}
\lhead{RAM FR1 report v 2.0}
\rhead{Date created: <DATE>}
\begin{document}
```

```
\section*{<SUBJECT> RAM FR1 Free Recall Report}

\textbf{\large Significant Electrodes}

<SIGNIFICANT_ELECTRODES>

\end{document}
```

The `<DATE>, <SUBJECT>, <SIGNIFICANT_ELECTRODES>` markup represents place holders where substitutions will take place. To do the substitutions we will use function `TextTemplateUtils.replace_template` from `TextTemplateUtils` package:

```
tex_session_template = 'my_documet.tex.tpl'
report_tex_file_name = 'my_documet.tex'

replace_dict = {
'<DATE>': datetime.date.today(),
'<SUBJECT>': 'R1059J',
'<SIGNIFICANT_ELECTRODES>': 'LR1,LT2',
}

TextTemplateUtils.replace_template
(
template_file_name=tex_session_template,
out_file_name=report_tex_file_name,
replace_dict=replace_dict
)
```

Here we are using `my_documet.tex.tpl` (template tex document shone above), tex template and will write results to `my_documet.tex` . We next create a Python dictionary where the keys correspond to markup we want to replace and the values correspond to values we will use to replace the markup. All that remains is to call `replace_template` and we will get the following tex document:

```
\documentclass[a4paper]{article}
\usepackage[usenames,dvipsnames,svgnames,table]{xcolor}
\usepackage{graphicx,multirow}
\usepackage{epstopdf}
\usepackage{subfigure,amsmath}
\usepackage{wrapfig}
\usepackage{longtable}
\usepackage{pdfpages}
\usepackage{mathtools}
\usepackage{array}
\usepackage{enumitem}
\usepackage{booktabs}
\usepackage{sidecap} \usepackage{soul}
\usepackage[small,bf,it]{caption}
\setlength\belowcaptionskip{2pt}

\addtolength{\oddsidemargin}{-.875in}
\addtolength{\evensidemargin}{-.875in}
\addtolength{\textwidth}{1.75in}
\addtolength{\topmargin}{-.75in}
\addtolength{\textheight}{1.75in}
```

```
\newcolumntype{C}[1]{>{\centering\let\newline\\\arraybackslash\hspace{0pt}}m{#1}}
\usepackage{fancyhdr}
\pagestyle{fancy}
\fancyhf{}
\lhead{RAM FR1 report v 2.0}
\rhead{Date created: 2015-12-22
\begin{document}


\section*{R1059J RAM FR1 Free Recall Report}

\textbf{\large Significant Electrodes}

LR1,LT2

\end{document}
```

As you can see keeping separate template documents with markup does not pollute main code and is general quite flexible solution.