Pydra

a new dataflow engine

Dorota Jarecka, Mathias Goncalves, Chris Markiewicz, Nicole Lo, Jakub Kaczmarzyk, Oscar Esteban, Satrajit Ghosh

MIT Stanford ReproNim NIH

Nipype: Neuroimaging in Python Pipelines and Interfaces

Acknowledgments

This was supported by NIH grants P41EB019936, R01EB020740. We thank the neuroimaging community for feedback during development.

What is Pydra?

- A lightweight, Py3.7+ dataflow engine for computational graph construction, manipulation, and distributed execution
- Designed as a **general-purpose engine** to support analytics in any scientific domain; created for *Nipype**
- Helps build reproducible, scalable, reusable, and fully automated, provenance tracked scientific workflows
- The power of Pydra lies in ease of workflow creation and execution for **complex multiparameter map-reduce operations**, and the use of global cache

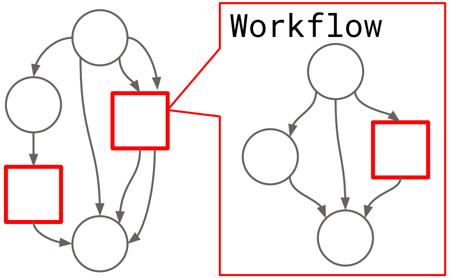
Repository and Tutorial:

- GitHub: https://github.com/nipype/pydra
- Tutorial: https://mybinder.org/v2/gh/nipype/pydra/master

* An open-source framework providing uniform Python interfaces, workflow construction, and execution for neuroimaging analyses.

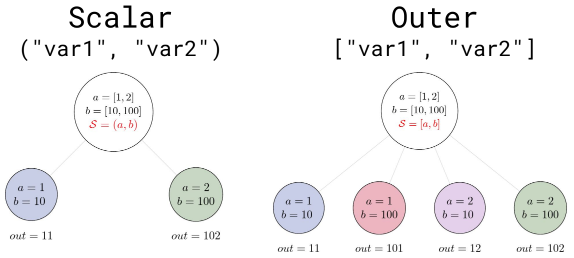
Pydra - Architecture, Features and Objects

Nested Workflows



Nested Splitters and Combiners

e.g. [("var1", "var2"), "var3"]



Task

- Workflow
- FunctionTask
- ShellCommandTask
  - ContainerTask
    - DockerTask
    - SingularityTask

Submitter

Worker

- ConcurrentFutures
- SLURM
- Dask (experimental)

Resource management

Multi location cache

- Writeable current cache: '/path/to/cache_dir'
- Readonly prior cache: ['/cache_dir1', '/cache_dir2']

Key Features

- Consistent API for Task and Workflow
- Splitting & combining semantics on Task/Workflow level
- Global cache support to reduce recomputation
- Support for execution of Tasks in containerized environments

Architecture

- Uses Python Standard Library (with few exceptions)
- Uses Concurrent Futures as the main executor (partial support for Slurm and Dask)
- Uses AsyncIO for asynchronous processes

Power of Pydra:

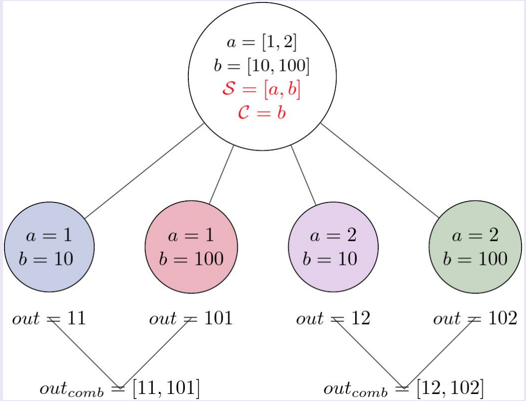
Splitter and Combiner

Task with an outer splitter *[a, b]* and a partial combiner *a*

```python
def main(a_list, b_list):
   out_comb_list = []
   for b in b_list:
       out_comb = []
       for a in a_list:
           out_comb.append(fun(a, b))
       out_comb_list.append(out_comb)
   return out_comb_list

main(a_list=[1, 2], b_list=[10, 100])
```

pydra

```python
task = fun(a=[1,2], b=[10,100])
           .split([“a”, “b”]).combine(“a”)
```



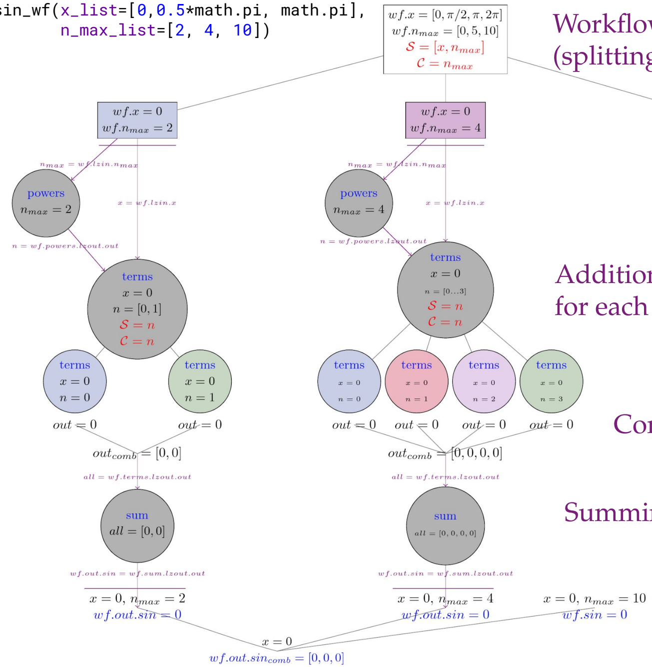
Workflow that calculates approximation of sine function

$$\sum_{n=0}^{n_{max}} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots$$

```python
def sin_wf(x_list, n_max_list):
   sin_comb_all = []
   for x in x_list:
       sin_comb = []
       for n_max in n_max_list:
           approx_terms = []
           for n in range(n_max):
                   approx_terms.append(term(x, n))
              sin_approx = sum(approx_terms)
       sin_comb.append(sin_approx)
       sin_comb_all.append(sin_comb)
   return sin_comb_all

sin_wf(x_list=[0,0.5*math.pi, math.pi],
       n_max_list=[2, 4, 10])
```

pydra

```python
wf = Workflow(name="wf", input_spec=["x", "n_max"])
wf.split(["x", "n_max"]).combine("n_max")
wf.add(range_fun(name="range",
                 n_max=wf.lzin.n_max))
wf.add(term(name="term", x=wf.lzin.x,
            n=wf.range.lzout.out)
            .split("n").combine("n"))
wf.add(summing(name="sum", terms=wf.term.lzout.out))
wf.set_output([("sin", wf.sum.lzout.out)])

wf.run(x=[0, 0.5 * math.pi, math.pi],
       n_max = [2, 4, 10])
```



Workflow with a splitter and a combiner (splitting over *x* and *n_max*)

Splitting input for the workflow

Additional splitter for Task in order to calculate term for each value of *n*

Combining all terms together

Summing all terms together for each value of *x* and *n_max*

$x = \pi/2$
$wf.out.sin_{comb} = [0.9248, 0.9998, 1.0]$
$x = \pi$
$wf.out.sin_{comb} = [-2.026, -0.075, -5e{-}10]$

Combining all approximations of *sin* for each value of *x*

Pydra - Scikit-learn Example - Bootstrapped model comparison

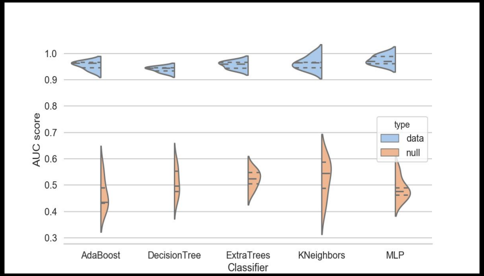
Decorate to create Pydra Tasks

```python
@pydra.mark.task # Convert python function to a Pydra Task object
@pydra.mark.annotate({"return": {"X": ty.Any, "Y": ty.Any, "groups":
ty.Any}})  # Annotate to create named outputs
def read_file(filename, x_indices=None, target_vars=None,
group='groups'):
    """Read and transform a CSV file to sklearn inputs"
    data = pd.read_csv(filename)
    X = data.iloc[:, x_indices]
    Y = data[target_vars]
    if group in data.keys():
        groups = data[:, [group]]
    else:
        groups = list(range(X.shape[0]))
    return X.values, Y.values, groups

@pydra.mark.task  # Convert python function to a Pydra Task object
@pydra.mark.annotate({"return": {"splits": ty.Any, "split_indices":
ty.Any}})  # Annotate to create named outputs
def gen_splits(n_splits, test_size, X, Y, groups=None, random_state=0):
    """Generate a set of train-test splits"
    from sklearn.model_selection import GroupShuffleSplit
    gss = GroupShuffleSplit(n_splits=n_splits, test_size=test_size,
                            random_state=random_state)
    train_test_splits = list(gss.split(X, Y, groups=groups))
    split_indices = list(range(n_splits))
    return train_test_splits, split_indices

@pydra.mark.task
@pydra.mark.annotate({"return": {"auc": ty.Any}})
def train_test_kernel(X, y, train_test_split, split_index, clf_info,
permute):
    """Train and test a classifier on actual or permuted labels"
    from sklearn.preprocessing import StandardScaler
    from sklearn.pipeline import Pipeline
    from sklearn.metrics import roc_auc_score
    mod = __import__(clf_info[0], fromlist=[clf_info[1]])
    clf = getattr(mod, clf_info[1])(**clf_info[2])
    if len(clf_info) > 3: # Run a GridSearch when param_grid available
        from sklearn.model_selection import GridSearchCV
        clf = GridSearchCV(clf, param_grid=clf_info[3])
    train_index, test_index = train_test_split[split_index]
    pipe = Pipeline([('std', StandardScaler()), (clf_info[1], clf)])
    y = y.ravel()
    if permute: # Run a generic permutation to create a null model
        pipe.fit(X[train_index], y[np.random.permutation(train_index)])
    else:
        pipe.fit(X[train_index], y[train_index])
    auc = roc_auc_score(y[test_index], pipe.predict(X[test_index]))
    return auc
```

Specify inputs

```python
clfs = [
 ('sklearn.ensemble', 'ExtraTreesClassifier',
        dict(n_estimators=100, class_weight='balanced')),
 ('sklearn.neural_network', 'MLPClassifier',
        dict(alpha=1, max_iter=1000)),
 ('sklearn.neighbors', 'KNeighborsClassifier', dict(),
        [{'n_neighbors': [3, 5, 7, 9, 11, 13, 15, 17, 19],
          'weights': ['uniform','distance']}]),
 ('sklearn.tree', 'DecisionTreeClassifier',
        dict(max_depth=5)),
 ('sklearn.ensemble', 'AdaBoostClassifier', dict())]
inputs = {"filename": os.path.abspath('breast_cancer.csv'),
          "x_indices": range(30), "target_vars": ('target',),
          "n_splits": 5, "test_size": 0.2,
          "clf_info": clfs,  "permute": [True, False]
          }
n_procs = 8
cache_dir = os.path.join(os.getcwd(), 'cache')
wf_cache_dir = os.path.join(os.getcwd(), 'cache-wf')
```

Output: Classifier performance of actual and null models over repeated independent train-test sampling



Cached runtimes script then workflow

```
Run 1: script
real      0m13.938s
user      0m58.722s
sys       0m9.089s

Run 2: workflow
reuse script cache

real      0m7.883s
user      0m22.457s
sys       0m6.156s
```

Use Pydra in script style:

```python
reader = read_file(filename=inputs["filename"],
                   x_indices=inputs["x_indices"],
                   target_vars=inputs["target_vars"],
                   cache_dir=cache_dir)
reader()  # Execute the task
res = reader.result() # Gather result
splitter = gen_splits(n_splits=inputs["n_splits"],
                      test_size=inputs["test_size"],
                      X=res.output.X, Y=res.output.Y,
                      groups=res.output.groups,
                      cache_dir=cache_dir)
splitter()
splitres = splitter.result()
clf_task = train_test_kernel(X=res.output.X, y=res.output.Y,
                   train_test_split=splitres.output.splits,
                   split_index=splitres.output.split_indices,
                         clf_info=inputs["clf_info"],
                         permute=inputs["permute"],
                         cache_dir=cache_dir)
# These two lines run the kernel over each
# classifier, train-test split, and with/without permutation
split_order = ['clf_info', 'permute', 'split_index']
clf_task.split(split_order).combine(split_order)

# Execute the final task in parallel using multiple procs
with pydra.Submitter(plugin="cf", n_procs=n_procs) as sub:
   sub(runnable=clf_task)
```

Use Pydra in Workflow style:

```python
# Encapsulate tasks in a Workflow reuse script output cache
wf = pydra.Workflow(name="ml_wf", input_spec=list(inputs.keys()),
               **inputs, cache_dir=wf_cache_dir, # workflow cache
                   cache_locations=[cache_dir]) # reuses script cache
# joint map over classifiers and permutation
wf.split(['clf_info', 'permute'])
wf.add(read_file(name="readcsv",                    # add task
                filename=wf.lzin.filename,          # connect workflow input
                x_indices=wf.lzin.x_indices,
                target_vars=wf.lzin.target_vars))
wf.add(gen_splits(name="gensplit",                  # add task
                 n_splits=wf.lzin.n_splits,         # connect workflow input
                 test_size=wf.lzin.test_size,
                 # connect lazy-eval outputs of previous task
                 X=wf.readcsv.lzout.X, Y=wf.readcsv.lzout.Y,
                 groups=wf.readcsv.lzout.groups))
wf.add(train_test_kernel(name="fit_clf", # use outputs from both tasks
                   X=wf.readcsv.lzout.X, y=wf.readcsv.lzout.Y,
                   train_test_split=wf.gensplit.lzout.splits,
                   split_index=wf.gensplit.lzout.split_indices,
                   clf_info=wf.lzin.clf_info, permute=wf.lzin.permute))
wf.fit_clf.split('split_index').combine('split_index') # Parallel spec
wf.set_output([("auc", wf.fit_clf.lzout.auc)]) # connect workflow output

# Execute the workflow in parallel using multiple processes
with pydra.Submitter(plugin="cf", n_procs=n_procs) as sub:
   sub(runnable=wf)
```