**Introduction**

Combine ML operations to flexible pipelines and processing graphs, which can be configured, trained, resampled, and tuned as any regular learner. The main purpose of a graph is to build combined preprocessing and modeling fitting pipelines that can be used as a Learner.

**PipeOp**

Flow operation with `train()` and `predict()` step.

- `$train()`: Training Data → PipeOp → Transformated Data
- `$predict()`: New Data → PipeOp → Transforated Data

Construction example: `pca = po("pca")`

- `Strain(input)`: Named list
- `Spredict(input)`: Named list
- `Gstate`: Learned parameters
- `Param_set`: See hyperparameters

**Popular PipeOps**

<table>
<thead>
<tr>
<th>Class</th>
<th>Key</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>PipeOpRemoveConstants</td>
<td>remove_constants</td>
<td>Repair Tasks</td>
</tr>
<tr>
<td>PipeOpScale</td>
<td>&quot;scale&quot;</td>
<td>Scale Features</td>
</tr>
<tr>
<td>PipeOpImputeMean</td>
<td>&quot;impute&quot;</td>
<td>Impute NAs</td>
</tr>
<tr>
<td>PipeOpFilter</td>
<td>&quot;filter&quot;</td>
<td>Feature Filter</td>
</tr>
<tr>
<td>PipeOpEncode</td>
<td>&quot;encode&quot;</td>
<td>Feature Encoding</td>
</tr>
<tr>
<td>PipeOpPCA</td>
<td>&quot;pca&quot;</td>
<td>PCA</td>
</tr>
<tr>
<td>PipeOpSelect</td>
<td>&quot;select&quot;</td>
<td>Restrict Columns</td>
</tr>
<tr>
<td>PipeOpCoApply</td>
<td>&quot;coapply&quot;</td>
<td>Transform Columns</td>
</tr>
<tr>
<td>PipeOpClassBalancing</td>
<td>&quot;classbalancing&quot;</td>
<td>Transform Imbalanced Data</td>
</tr>
<tr>
<td>PipeOpLearn</td>
<td>&quot;learn&quot;</td>
<td>Use Learner</td>
</tr>
<tr>
<td>PipeOpLearnCV</td>
<td>&quot;learn_cv&quot;</td>
<td>Crossval Learner</td>
</tr>
<tr>
<td>PipeOpMutate</td>
<td>&quot;mutate&quot;</td>
<td>Feature Engineering</td>
</tr>
<tr>
<td>PipeOpChunk</td>
<td>&quot;chunk&quot;</td>
<td>Split Data</td>
</tr>
<tr>
<td>PipeOpSubSample</td>
<td>&quot;subsample&quot;</td>
<td>Subsample Rows</td>
</tr>
<tr>
<td>PipeOpFeatureUnion</td>
<td>&quot;featureunion&quot;</td>
<td>Combine Features</td>
</tr>
<tr>
<td>PipeOpFlipFactors</td>
<td>&quot;flipfactors&quot;</td>
<td>Handle Unknown Levels</td>
</tr>
<tr>
<td>PipeOpNOP</td>
<td>&quot;nop&quot;</td>
<td>Do Nothing</td>
</tr>
</tbody>
</table>

Full list: `as.data.table(mlr_pipes)

**Graph**

Connects PipeOps with edges to control data flow during training and prediction. Input is sent to sources (no in-edges), output is read from sinks (no out-edges). Important methods and slots:

- `Display print()`: Display PipeOps or gr $pipeops Named list of all contained PipeOps.
- `Accessing PipeOps`: `gr$pipeops`

**GraphConstruction**

The pipe operator takes either a PipeOp or a Graph on each of its sides and connects all left-hand outputs to the right-hand inputs. For full control, connect PipeOps explicitly.

```
gr = GraphLearner$new()
gr$add_pipeop(po("pca"))
gr$add_pipeop(lrn("classif.rpart"))
gr$add_edge("pca", "classif.rpart")
```

**GraphLearner**

GraphLearner behave like Learner and enable all mlr3 operations to execute pipelines and processing graphs, which can be configured, trained, resampled, and tuned as any regular learner. The main purpose of a graph is to build combined preprocessing and modeling fitting pipelines that can be used as a Learner.

- `$train()`: Training Data → GraphLearner → Training Data
- `$predict()`: New Data → GraphLearner → Prediction
- `$model`: Learned parameters
- `$state`: Learned parameters

For pipe: `gr$train()`, `gr$predict()` or a Graph:

```
graphLearner = GraphLearner$new()
graphLearner$train(task, grl, rsmp, folds = 3)
graphLearner$predict(task, grl, rsmp)
```

**Debugging and Intermediate Results**

- `keep_results` for PipeOps.
- `store_intermediate` for PipeOps.
- `store_intermediate` for PipeOps.
- `Returns intermediate result of train()` and `predict()`.
- `e.g. modified task resulted by encode pipeop`.

```
gr$state
```

**Hyperparameters**

For POS: Exactly as in a Learner.

```
enc = po("encode")
enc$param_set_values = list(method = "one-hot")
enc$predict()  # Name values
enc$param_vals = list(method = "one-hot")
```

For Graph / GraphLearner: All HPs are collected in a global ParamSet stored in `$param_set`. IDs are prefixed with the respective PipeOp id.

**Tuning**

Can jointly tune any Pipeline.

**Feature Engineering**

PipeOpMutate adds new features. This works by providing expressions in a `list`.

```
Example

task = mlr_task["iris"]
s = mlr_formatter(task)  #
lrn = mlr_learner["classif.rpart"]
s$param_set = list(tagging = "one-hot")
graphLearner$mutate(task, lrn, s$param_set)
```

**Logging**

```
log = logger("mlr3pipelines")
log$set_threshold(<level>)
```

Change log-level only for mlr3pipelines.

**Nonlinear Graphs**

- `gunion()` creates a new Graph containing n copies of the input (PipeOp or Graph).

- `pipeline_greplicate()` creates a new Graph containing n copies of the input (PipeOp or Graph).

**Linear Graphs**

Concatenate POS with `>>`:

```
Example

task = mlr_task["iris"]
inst = mlr_instance(task, sample = NULL)
msr = mlr_measure(task, name = "classif.ce")

tuner = mlr_tuner[c("random_search", "grid_search")]
tuner$param_set = ParamSet$new(levels = list(eta = c(0, 0.05)), rsmp)

t = mlr_train(tuner, inst, msr)
```

**Branching**

Controls the path execution. Only one branch can be active. Which one is controlled by a hyperparameter. Unbranching ends the forking.

```
Example

t = mlr_task["penguins"]
s = mlr_formatter(t)  #
inst = mlr_instance(t)
msr = mlr_measure(t, name = "classif.ce")

tuner = mlr_tuner[c("random_search", "grid_search")]
tuner$param_set = ParamSet$new(levels = list(eta = c(0, 0.05)), rsmp)

t = mlr_train(tuner, inst, msr)
```

**Example**

```
po = PipeOp$new()
pop_a = PipeOpFeatureUnion$new()  #
pop_a$add_pipeop(po("pca"))  #
```

Cheatdown on GitHub