FTPVL

Aug 18, 2020

Getting Started

FPGA Tool Performance Visualization Library (FTPVL) is a library for simplifying the data collection, analysis, and visualization of performance metrics for SymbiFlow development. Although it was made with SymbiFlow in mind, it is highly extensible for future integration with other software.

CHAPTER 1

Getting Started

Learn how to get up and running with FTPVL.

1.1 First Steps

FTPVL works best when installed in an interactive Python environment, such as a Jupyter notebook. Many of the examples in the documentation will be accessible through notebooks hosted on Google Colaboratory.

You can follow along with the steps below by running the cells in this colab notebook.

1.1.1 Installing FTPVL

Note: It's recommended to set up a virtual environment before installing FTPVL to prevent future issues with system-wide packages.

Let's get started by installing FTPVL. The easiest way is to download it using PyPi:

```
pip install ftpvl
```

In a Python notebook, you can also perform command-line operations by writing the the command into a cell prefixed with an `!`:

```
!pip install ftpvl
```

Now, let's import the classes that we will need to complete this tutorial:

```
from ftpvl.fetchers import HydraFetcher
from ftpvl.processors import *
from ftpvl.styles import ColorMapStyle
from ftpvl.visualizers import SingleTableVisualizer
```

1.1.2 Fetching Data from Hydra

Fetchers in FTPVL are responsible for ingesting data from a source and performing simple pre-processing to standardize the output. This results in the creation of an *Evaluation* instance, which stores the fetched test results for a single execution of the testing suite.

The most common place to fetch test results is from Hydra. To accomplish this, we use the `HydraFetcher`.

We first must specify a set of mappings between the JSON object properties provided by Hydra and the desired metric name. This metric name will be used to reference the field for all future processing.

Note: Hydra provides test results as nested JSON objects. This is decoded by `HydraFetcher` and *flattened* to make it easier to reference nested performance metrics using a string. You can reference a nested metric by delimiting each metric to index into with a `.`. For example, the result `{"a": {"b": "c"}}` is flattened to `{"a.b": "c"}`.

```
# mappings from Hydra JSON object properties to metric names
df_mappings = {
    "project": "project",
    "device": "device",
    "resources.BRAM": "bram",
    "resources.CARRY": "carry",
    "resources.DFF": "dff",
    "resources.IOB": "iob",
    "resources.LUT": "lut",
    "resources.PLL": "pll",
    "runtime.synthesis": "synthesis",
    "runtime.packing": "pack",
    "runtime.placement": "place",
    "runtime.routing": "route",
    "runtime.fasm": "fasm",
    "runtime.bitstream": "bitstream",
    "runtime.total": "total"
}
```

Next, we can specify the clock names to use when parsing the Hydra JSON results. Since different tests might contain one or more clock frequencies, we specify a ranked list of clock frequency symbols, using the first matching entry as the clock frequency in our analysis.

```
# the ordered list of clock names to reference
hydra_clock_names = ["clk", "sys_clk", "clk_i"]
```

We can now use those variables as parameters for the `HydraFetcher`. Specify the desired project ID and jobset name from `hydra.vtr.tools` that will be used when fetching. This information can be found through the web interface. To get the latest evaluation, set `eval_num` to `0`. We set `eval_num` to `2` in the example below since it is the latest evaluation (as of this writing) that passes at least one test case.

> **Warning:** The `eval_num` parameter must reference an evaluation with at least one passing test. Without this, HydraFetcher will raise a `ValueError`. You can determine this by using the web interface to ensure that the selected evaluation number has at least one passing test.

```
eval1 = HydraFetcher(
    project="dusty",
    jobset="fpga-tool-perf",
```

(continues on next page)

(continued from previous page)

```
    eval_num=2,
    mapping=df_mappings,
    hydra_clock_names=hydra_clock_names
).get_evaluation()
```

1.1.3 Processing Data

After fetching the data, we will need to process the raw data to extract meaningful results that can be visualized. FTPVL performs processing through the use of a processing pipeline, which applies consecutive transformations to arrive at the desired output.

The pipeline is constructed as a list of *Processors*, which are the primitive transformations implemented in FTPVL.

The `StandardizeTypes` processor casts each metric in the test results to a certain type, which prevents type errors during future transformations. We specify a dictionary mapping the metric names to the desired type:

```
# specify the types to cast to
df_types = {
    "project": str,
    "device": str,
    "toolchain": str,
    "freq": float,
    "bram": int,
    "carry": int,
    "dff": int,
    "iob": int,
    "lut": int,
    "pll": int,
    "synthesis": float,
    "pack": float,
    "place": float,
    "route": float,
    "fasm": float,
    "bitstream": float,
    "total": float
}
```

The `ExpandColumn` processor adds additional metrics to the Evaluation by reading the value of a pre-existing metric and adding new metrics based on a mapping.

In this case, we want to be able to sort by the synthesis tool and place-and-route tool for each test case, but those are not specified by Hydra. Instead, we can read the pre-existing `toolchain` value for each test case, and write a `synthesis_tool` and `pr_tool` metric based on the toolchain.

```
# specify how to convert toolchains to synthesis_tool/pr_tool
toolchain_map = {
    'vpr': ('yosys', 'vpr'),
    'vpr-fasm2bels': ('yosys', 'vpr'),
    'yosys-vivado': ('yosys', 'vivado'),
    'vivado': ('vivado', 'vivado'),
    'nextpnr-ice40': ('yosys', 'nextpnr'),
    'nextpnr-xilinx': ('yosys', 'nextpnr'),
    'nextpnr-xilinx-fasm2bels': ('yosys', 'nextpnr')
}
```

Now, we construct the actual pipeline for processing the data. You can read the specifications of each processor in the *Processors API* reference.

```
# define the pipeline to process the evaluation
processing_pipeline = [
    StandardizeTypes(df_types),
    CleanDuplicates(
        duplicate_col_names=["project", "toolchain"],
        sort_col_names=["freq"]),
    AddNormalizedColumn(
        groupby="project",
        input_col_name="freq",
        output_col_name="normalized_max_freq"),
    ExpandColumn(
        input_col_name="toolchain",
        output_col_names=("synthesis_tool", "pr_tool"),
        mapping=toolchain_map),
    Reindex(["project", "synthesis_tool", "pr_tool", "toolchain"])
    SortIndex(["project", "synthesis_tool"])
]
```

Finally, we can apply the processing pipeline to the evaluation by using the `process()` method.

```
eval1 = eval1.process(processing_pipeline)
```

1.1.4 Styling

Now that the Evaluation has been processed, we can add styling so that important information stands out in the final visualization. This is achieved through a special type of Processor called *Styles*.

Styles are also run in a processing pipeline, but they always output CSS strings. We will use the `ColorMapStyle` to color results that are better or worse than a baseline result.

First, we specify which columns are styled, and the direction which they should be optimized. Some columns are better if the value is minimized (such as compilation times) while others are better if the value is maximized (such as frequency).

```
# generate styling
styled_columns = {
    "bram": 1, # optimize by minimizing
    "carry": 1,
    "dff": 1,
    "iob": 1,
    "lut": 1,
    "synthesis": 1,
    "pack": 1,
    "place": 1,
    "route": 1,
    "fasm": 1,
    "bitstream": 1,
    "total": 1,
    "freq": -1, # optimize by maximizing
    "normalized_max_freq": -1
}
```

Next, we generate a Matplotlib colormap using `seaborn`, which will be used to generate a diverging color palette for values that are either better or worse than the baseline. If it is better, the cell will be greener. If worse, the cell will be

redder.

```
import seaborn as sns
cmap = sns.diverging_palette(180, 0, s=75, l=75, sep=100, as_cmap=True)
```

Finally, we can create the styled evaluation by processing the evaluation above with the `NormalizeAround` processor to calculate which values are better or worse than the baseline, followed by the `ColorMapStyle` style to generate the CSS styles using the colormap.

```
styled_eval = eval1.process([
    NormalizeAround(
        styled_columns,
        group_by="project",
        idx_name="synthesis_tool",
        idx_value="vivado"),
    ColorMapStyle(cmap)
])
```

1.1.5 Visualization

Our last step is to display the processed evaluation and its style. We first add some custom static styles that do not depend on the input data. These are used for adding styles on hover and adding borders to help visually separate the test results.

```
custom_styles = [
    dict(selector="tr:hover", props=[("background-color", "#99ddff")]),
    dict(selector=".level0", props=[("border-bottom", "1px solid black")]),
    dict(selector=".level1", props=[("border-bottom", "1px solid black")]),
    dict(selector=".level2", props=[("border-bottom", "1px solid black")]),
    dict(selector=".level3", props=[("border-bottom", "1px solid black")])
]
```

Then, we use the *Visualizers* in FTPVL to generate an IPython-compatible visualization that can be displayed.

```
vis = SingleTableVisualizer(
    eval1,
    styled_eval,
    version_info=True,
    custom_styles=custom_styles
)
display(vis.get_visualization())
```

project	synthesis_tool	pr_tool	toolchain	bram	carry	dff	iob	lut	pll	synthesis	pack	place	route	fasm	bitstream	total	freq	normalized_max_freq
blinky	vivado	vivado	vivado	0	6	32	13	20	0	22.00	-	-	14.00	-	8.00	114.25	481.70	1.00
	yosys	nextpnr	nextpnr-xilinx	0	8	30	13	79	0	5.81	-	0.15	3.47	-	1.59	13.57	260.35	0.54
			nextpnr-xilinx-fasm2bels	0	8	30	13	79	0	5.65	-	0.15	3.56	-	1.53	229.05	280.74	0.58
		vivado	yosys-vivado	0	8	30	13	15	0	5.26	-	-	14.00	-	8.00	41.26	359.45	0.75
		vpr	vpr	0	8	30	13	15	0	7.55	1.34	1.34	1.41	26.22	1.76	78.00	241.60	0.50
			vpr-fasm2bels	0	8	30	13	15	0	7.33	1.31	1.35	1.38	26.43	1.67	376.98	242.13	0.50
hamsternz-hdmi	vivado	vivado	vivado	11	252	1895	34	2518	2	56.00	-	28.00	56.00	-	16.00	240.62	121.33	1.00
ibex	vivado	vivado	vivado	32	56	1882	6	3093	1	220.00	-	10.00	17.00	-	9.00	344.48	49.56	1.00
	yosys	vpr	vpr	0	71	2050	6	7125	1	137.18	25.77	1.77	82.12	42.72	107.06	476.62	-	-
litex-linux	vivado	vivado	vivado	81	260	4883	89	6207	1	138.00	-	24.00	26.00	-	13.00	305.15	87.83	0.48
	yosys	vivado	yosys-vivado	89	319	5177	92	7494	1	107.53	-	43.00	84.00	-	13.00	277.53	182.75	1.00
		vpr	vpr	35	325	5267	89	10790	1	160.61	114.51	100.49	167.94	65.00	159.15	914.74	56.43	0.31
			vpr-fasm2bels	35	325	5267	89	10790	1	144.83	105.69	92.41	156.85	60.88	146.89	2581.96	56.68	0.31
murax	vivado	vivado	vivado	6	50	1036	35	967	0	42.00	-	-	14.00	-	8.00	135.01	127.15	0.98
	yosys	vivado	yosys-vivado	4	57	1085	35	1044	0	19.82	-	-	16.00	-	9.00	59.82	129.13	1.00
		vpr	vpr	4	57	1082	35	1378	0	28.47	22.67	6.90	10.53	29.25	23.76	167.52	79.05	0.61

1.2 Examples

You can take a look at some other examples of FTPVL at work in Google Colab notebooks.

- Using HydraFetcher and Processors
- Styling Tables with SingleTableVisualizer
- Comparing Two Different Evaluations

First Steps The first steps for using FTPVL.

Examples Look at some real-world examples to see FTPVL at work!

CHAPTER 2

Overview

Dive into the main features of FTPVL.

2.1 Evaluation

2.1.1 Overview

Evaluations store the test results from a single execution of the test suite. They are constructed either using *Fetchers* or manually by specifying a Pandas dataframe and evaluation ID.

2.1.2 Processing

Evaluations can be processed by using the `process()` method. Its only parameter is a list of *Processors* that will be used to transform the evaluation one-at-a-time.

`Evaluation.`**`process`**(*pipeline: List[Processor]*) → Evaluation

Executes each processor in the pipeline and returns a new Evaluation.

Parameters **`pipeline`** (*a list of Processors to process the Evaluation in order*) –

Returns an Evaluation instance that was processed by the pipeline

2.1.3 Extracting the internal dataframe

Evaluations store the test results internally using a Pandas dataframe. You can retrieve this dataframe by using the `get_df()` method. This can be used to manipulate the underlying data without needing to use the built-in FTPVL processors.

Note: `get_df()` returns a defensive copy of the dataframe, so any mutations to the returned dataframe will not be reflected in the original Evaluation. Instead, you must instantiate a new `Evaluation` by passing in the dataframe and evaluation ID.

`Evaluation.`**`get_df`**`()` → pandas.core.frame.DataFrame
Returns a copy of the Pandas DataFrame that represents the evaluation

Example

```
>>> eval1 = Evaluation(pd.DataFrame([{"a": 1, "b": 2}, {"a": 4, "b": 5}]), eval_id=1)
>>> df = eval1.get_df() # extract Pandas dataframe
>>> df
    a   b
0   1   2
1   4   5
>>> df["c"] = [3, 6] # add a new column
>>> eval2 = Evaluation(df, eval_id=eval1.get_eval_id()) # create new evaluation
>>> eval2.get_df()
    a   b   c
0   1   2   3
1   4   5   6
```

2.2 Fetchers

2.2.1 Overview

Fetchers serve as a convenient way to ingest information from a source and create an *Evaluation*. Sources can either be local or over the network.

2.2.2 Fetching from Hydra

You can fetch from Hydra by using the *HydraFetcher* fetcher. Its functionality is explained in the *Fetching Data from Hydra* section of the *First Steps* guide.

2.2.3 Fetching from a JSON dataframe

You can also fetch from a properly-formatted JSON dataframe by using the *JSONFetcher* fetcher and specifying the path to the JSON file in the parameter. This is useful if some pre-processing needs to be performed, or the test runner is able to output a dataframe as a build artifact.

Most commonly, this fetcher can import JSON-encoded dataframes if they are exported from a separate Pandas dataframe. Learn about exporting as a JSON file from Pandas.

2.2.4 Getting the Evaluation

To get the *Evaluation* object from the Fetcher, call the `get_evaluation()` method.

`Fetcher.`**`get_evaluation`**`()` → ftpvl.evaluation.Evaluation
Returns an Evaluation that represents the fetched data.

2.3 Processors

2.3.1 Overview

Processors transform an evaluation to make it easier to draw conclusions from the data. They are useful in converting test data fetched using a *Fetchers* into the desired format for visualization.

One example of a processor is `Normalize`, which normalizes all specified test metrics around zero to improve the understandability of the results.

2.3.2 Processing Pipelines

Processors can be chained together to form a *processing pipeline*, which is a list of processors that are used one after the other to transform an Evaluation.

Here is an example of the processing pipeline that is used in the *First Steps* guide:

```
processing_pipeline = [
    StandardizeTypes(df_types),
    CleanDuplicates(
        duplicate_col_names=["project", "toolchain"],
        sort_col_names=["freq"]),
    AddNormalizedColumn(
        groupby="project",
        input_col_name="freq",
        output_col_name="normalized_max_freq"),
    ExpandColumn(
        input_col_name="toolchain",
        output_col_names=("synthesis_tool", "pr_tool"),
        mapping=toolchain_map),
    Reindex(["project", "synthesis_tool", "pr_tool", "toolchain"]),
    SortIndex(["project", "synthesis_tool"])
]
```

2.4 Styles

2.4.1 Overview

Styles are a special subclass of *Processors* that transform the values inside an *Evaluation* into CSS styles. The output of a Style can then be used with certain *Visualizers* to add color to the final display.

2.4.2 Using Colormaps

The *ColorMapStyle* style takes a Matplotlib Colormap as a parameter, which allows for the background color of each cell to be dependent on the value inside it. Since *Colormaps* expect the input value to be between 0 and 1, we usually have a processor in the pipeline before applying the Style. This is demonstrated in the *Styling* section of the *First Steps* guide.

2.5 Visualizers

2.5.1 Overview

Visualizers bring the data to life by creating a displayable representation of one or more *Evaluation*.

2.5.2 Version Info

You can choose to show the version info of each test result by setting the `version_info` parameter to True (which is False by default). This results in the display of the version info that is provided by Hydra.

2.5.3 Displaying the Visualization

After instantiating the desired visualizer, you can call the `get_visualization()` method to return an object that can be displayed in an IPython notebook by calling the `display()` function (documentation).

`ftpvl.visualizers.Visualizer.`**`get_visualization`**(*self*)
Returns a displayable object which can be displayed by calling display() in an interactive Python environment.

Evaluation Main data structure used for manipulation.

Fetchers Ingest data from Hydra or a JSON dataframe.

Processors Perform processing on the results using a processing pipeline.

Styles Generate styling for custom visualizations.

Visualizers Visualize the processed data.

CHAPTER 3

Reference

3.1 Core API

3.1.1 Evaluation API

class ftpvl.evaluation.**Evaluation**(*df: pandas.core.frame.DataFrame*, *eval_id: int = None*)

A collection of test results from a single evaluation of a piece of software on one or more test cases.

Parameters

- **df** (*pd.DataFrame*) – The dataframe that contains the test results of the given evaluation, rows for each test case and columns for each recorded metric
- **eval_id** (*int, optional*) – The ID number of the evaluation, by default None

get_copy() → ftpvl.evaluation.Evaluation

Returns a deep copy of the Evaluation instance

get_df() → pandas.core.frame.DataFrame

Returns a copy of the Pandas DataFrame that represents the evaluation

get_eval_id() → Optional[int]

Returns the ID number of the evaluation if specified, otherwise None

process(*pipeline: List[Processor]*) → Evaluation

Executes each processor in the pipeline and returns a new Evaluation.

Parameters **pipeline** (*a list of Processors to process the Evaluation in order*) –

Returns an Evaluation instance that was processed by the pipeline

3.1.2 Fetchers API

Fetchers are responsible for ingesting and standardizing data for future processing.

HydraFetcher

class ftpvl.fetchers.**HydraFetcher**(*project: str*, *jobset: str*, *eval_num: int = 0*, *absolute_eval_num: bool = False*, *mapping: dict = None*, *hydra_clock_names: list = None*)

Represents a downloader and preprocessor of test results from *hydra.vtr.tools*.

Parameters

- **project** (*str*) – The project name to use when fetching from Hydra
- **jobset** (*str*) – The jobset name to use when fetching from Hydra
- **eval_num** (*int, optional*) – An integer that specifies the evaluation to download. Functionality differs depending on whether *absolute_eval_num* is True, by default 0
- **absolute_eval_num** (*bool, optional*) – Flag that specifies if the eval_num is an absolute identifier instead of a relative identifier. If True, the fetcher will find an evaluation with the exact ID in *eval_num*. If False, *eval_num* should be a non-negative integer with *0* being the latest evaluation and *1* being the second latest evaluation, etc. By default False.
- **mapping** (*dict, optional*) – A dictionary mapping input column names to output column names, if needed for remapping, by default None
- **hydra_clock_names** (*list, optional*) – An optional ordered list of strings used in finding the actual frequency for each build result, by default None

get_evaluation() → ftpvl.evaluation.Evaluation

Returns an Evaluation that represents the fetched data.

JSONFetcher

class ftpvl.fetchers.**JSONFetcher**(*path: str*, *mapping: dict = None*)

Represents a loader and preprocessor of test results from a JSON file.

Parameters

- **path** (*str*) – A string file path pointing to the dataframe JSON file.
- **mapping** (*dict, optional*) – An optional dictionary mapping input column names to output column names., by default None

get_evaluation() → ftpvl.evaluation.Evaluation

Returns an Evaluation that represents the fetched data.

3.1.3 Processors API

Processors transform Evaluations to be more useful when visualized.

class ftpvl.processors.**AddNormalizedColumn**(*groupby: str*, *input_col_name: str*, *output_col_name: str*, *direction: ftpvl.processors.Direction = <Direction.MAXIMIZE: 1>*)

Processor that groups rows by a column, finds the best value of the specified column, and adds a new column with the normalized values of the row compared to the best value.

The best value is specified by the direction parameter. If the direction is *1*, then the best value is the maximum. If direction is *-1*, then the best value is the minimum.

Parameters

- **groupby** (*str*) – the column to group by
- **input_col_name** (*str*) – the input column to normalize
- **output_col_name** (*str*) – the column to write the normalized values to
- **direction** (Direction) – specifies how to find the 'best' value to normalize against. By default MAXIMIZE, all values will be compared to the max value of the input column.

class ftpvl.processors.**CleanDuplicates**(*duplicate_col_names: List[str], sort_col_names: List[str] = None, reverse_sort: bool = False*)

Processor that outputs a new dataframe that removes duplicate rows. Optionally can sort the dataframe before removing duplicates.

Two rows are considered duplicates if and only if all values specified in duplicate_col_names matches.

By default, the first instance of a duplicate is retained, and all others are removed. You can optionally specify columns to sort by and which way to sort, which provides fine-grained control over which rows are removed.

Parameters

- **duplicate_col_names** (*List[str]*) – column names to use when finding duplicates
- **sort_col_names** (*List[str], optional*) – column names to sort by, by default None
- **reverse_sort** (*bool, optional*) – sort in ascending order, by default False

class ftpvl.processors.**ExpandColumn**(*input_col_name: str, output_col_names: List[str], mapping: Dict[str, List[str]]*)

Processor that turns one column into more than one column by mapping values of a column to multiple values.

Parameters

- **input_col_name** (*str*) – the column name to map from
- **output_col_names** (*List[str]*) – the column names to map to
- **mapping** (*Dict[str, List[str]]*) – a dictionary mapping a value to a list of values. The processor will look at the value at input_col_name, use it as the key to index into mapping, and get the corresponding list of strings that will used as the values for the new metrics with names in output_col_names.

class ftpvl.processors.**MinusOne**

Processor that returns the input Evaluation by subtracting one from every data value.

This processor is useful for testing the functionality of processors on Evaluations.

class ftpvl.processors.**Normalize**(*normalize_direction: Dict[str, ftpvl.processors.Direction]*)

Processor that normalizes all specified values in an Evaluation by column around zero.

All normalized values are between 0 and 1, with 0.5 being the baseline. Therefore, a value of zero is mapped to 0.5, positive values are mapped to values > 0.5, and negative values are mapped to values < 0.5.

This is useful in creating styles for evaluations that have already performed calculations to compare multiple evaluations. For example, you can subtract one evaluation from another, then apply this processor before styling.

Parameters **normalize_direction** (*Dict[str, Direction]*) – a dictionary mapping column names to the optimization direction of the column. Used to determine if increases or decreases to baseline are perceived to be 'better'.

class ftpvl.processors.**NormalizeAround**(*normalize_direction: Dict[str, ftpvl.processors.Direction], group_by: str, idx_name: str, idx_value: str*)

Processor that normalizes all specified values in an Evaluation around a baseline that is chosen based on a specified index name and value.

All normalized values are between 0 and 1, with 0.5 being the baseline. This is useful in creating styles that show relative differences between each row and the baseline.

Parameters

- **normalize_direction** (*Dict[str, Direction]*) – a dictionary mapping column names to the optimization direction of the column. Used to determine if increases or decreases to baseline are perceived to be 'better'.
- **group_by** (*str*) – the column name used to group results before finding the baseline of the group and normalizing
- **idx_name** (*str*) – the name of the index used to find the baseline result. The baseline result will become the baseline which all other grouped results will be normalized by
- **idx_value** (*str*) – the value of the baseline result at idx_name

class ftpvl.processors.**Reindex**(*reindex_names: List[str]*)

Processor that reassigns current columns as indices for improved visualization.

Reindexing is useful for grouping similar results in the final visualization, since each row is identified by a (usually-unique) value in the index. Indices are always the first columns, which improves readability of the final visualization.

Parameters **reindex_names** (*List[str]*) – A list of column names to reindex

class ftpvl.processors.**SortIndex**(*sort_names: List[str]*)

Processor that sorts an evaluation by indices.

Parameters **sort_names** (*List[str]*) – a list of index names to sort by

class ftpvl.processors.**StandardizeTypes**(*types: dict*)

Processor that casts metrics in an Evaluation to the specified type.

The type of each metric in an Evaluation is inferred after fetching. This processor accepts a dictionary of types and casts the Evaluation to those types.

Parameters **types** (*dict*) – A mapping from column names to types

class ftpvl.processors.**RelativeDiff**(*a: ftpvl.evaluation.Evaluation*)

Processor that outputs the relative difference between evaluation A and B.

All numeric metrics will be compared, and all others will not be included in the output. B is compared to A, where the output is greater than 0 if B is greater than A, and less than 0 otherwise.

The calculation performed is (B - A) / A, where B is the evaluation that this processor is being applied to and A is the evaluation passed as a parameter.

Parameters **a** (Evaluation) – The evaluation to use when comparing against the Evaluation that is being processed. Corresponds to evaluation A in the description.

Examples

```
>>> a = Evaluation(pd.DataFrame(
... data=[
...     {"x": 1, "y": 5},
...     {"x": 4, "y": 10}
... ]))
>>> b = Evaluation(pd.DataFrame(
... data=[
...     {"x": 2, "y": 20},
```

(continues on next page)

(continued from previous page)

```
...     {"x": 2, "y": 2}
... ]))
>>> b.process([RelativeDiff(a)]).get_df()
     x    y
0  1.0  3.0
1 -0.5 -0.8
```

class ftpvl.processors.**FilterByIndex**(*index_name: str*, *index_value: Any*)

Processor that filters an Evaluation by matching a specified index value after indexing.

This is best used in a processing pipeline after the Reindex processor. For filtering an evaluation based on metrics (which is not an index), use the FilterByMetric processor.

Parameters

- **index_name** (*str*) – the name of the index to use when filtering
- **index_value** (*Any*) – the value to compare with

Examples

```
>>> a = Evaluation(pd.DataFrame(
... data=[
...     {"x": 1, "y": 5},
...     {"x": 4, "y": 10}
... ],
... index=pd.Index(["a", "b"], name="key")))
>>> a.process([FilterByIndex("key", "a")]).get_df()
    x    y
key
a   1    5
```

class ftpvl.processors.**Aggregate**(*func: Callable[[pandas.core.series.Series], Union[int, float]]*)

Processor that allows you to aggregate all the numeric fields of an Evaluation using a specified function.

This acts as a superclass for specific aggregator implementations, such as GeomeanAggregate. It can also be used for custom aggregations, by supplying an aggregator function to the constructor.

Parameters **func** (*Callable[[pd.Series], Union[int, float]]*) – a function that takes a Pandas Series and aggregates it into a single number, possibly a NaN value

Examples

```
>>> a = Evaluation(pd.DataFrame(
... data=[
...     {"x": 1, "y": 5},
...     {"x": 4, "y": 10}
... ]))
>>> a.process([Aggregate(lambda x: x.sum())]).get_df()
    x    y
0   5    15
```

class ftpvl.processors.**GeomeanAggregate**

Processor that aggregates an entire Evaluation by finding the geometric mean of each numeric metric.

Subclass of Aggregate class.

Examples

```
>>> a = Evaluation(pd.DataFrame(
... data=[
...     {"x": 1, "y": 8},
...     {"x": 4, "y": 8}
... ]))
>>> a.process([GeomeanAggregate()).get_df()
    x    y
0   2    8
```

class ftpvl.processors.**CompareToFirst**(*normalize_direction: Dict[str, ftpvl.processors.Direction], suffix: str = '.relative'*)

Processor that compares numeric rows in an evaluation to the first row by adding columns that specify the relative difference between the first row and all other rows.

You can specify the direction that improvements should be outputted. For example, a change from *100* to *50* may be a 2x change if the objective is minimization, while it may be a 0.5x change if the objective is maximization.

Parameters

- **normalize_direction** (*Dict[str, Direction]*) – a dictionary mapping column names to the optimization direction of the column. Used to determine if increases or decreases to baseline are perceived to be 'better'.
- **suffix** (*str*) – the suffix to use when creating new columns that contain the relative comparison to the first row, by default ".relative"

Examples

```
>>> a = Evaluation(pd.DataFrame(
... data=[
...     {"x": 1, "y": 8},
...     {"x": 4, "y": 8}
... ]))
>>> direction = {"x": Direction.MAXIMIZE, "y": Direction.MAXIMIZE}
>>> a.process([CompareToFirst(direction, suffix=".diff")).get_df()
    x   x.diff  y   y.diff
0   1   1.00    8   1.0
1   4   4.00    8   1.0
```

```
>>> a = Evaluation(pd.DataFrame(
... data=[
...     {"x": 1, "y": 8},
...     {"x": 4, "y": 8}
... ]))
>>> direction = {"x": Direction.MINIMIZE, "y": Direction.MINIMIZE}
>>> a.process([CompareToFirst(direction, suffix=".diff")).get_df()
    x   x.diff  y   y.diff
0   1   1.00    8   1.0
1   4   0.25    8   1.0
```

3.1.4 Styles API

Styles are special processors that transform an evaluation into CSS styles.

ColorMapStyle

class ftpvl.styles.**ColorMapStyle**(*cmap: matplotlib.colors.Colormap*)

Represents a processor that uses a matplotlib Colormap to create a styled evaluation where the background of a cell is specified by the value in the cell evaluated by the colormap.

Values in the input are expected to be either a float between 0 and 1 (inclusive) or a empty string.

Parameters **cmap** (*Colormap*) – the colormap to use when transforming input Evaluation to CSS styles.

3.1.5 Visualizers API

Visualizers are used for displaying the results to the user in an IPython notebook.

class ftpvl.visualizers.**DebugVisualizer**(*evaluation: ftpvl.evaluation.Evaluation*, *version_info: bool = False*, *custom_styles: List[dict] = None*, *column_order: List[str] = None*)

Represents a visualizer that will print the given Evaluation, possibly with version information.

Useful for debugging.

Parameters

- **evaluation** (Evaluation) – the Evaluation to display
- **version_info** (*bool, optional*) – Flag to display version information from the build results in the final visualization, by default False
- **custom_styles** (*List[dict], optional*) – Specify additional styling for the final visualzation. See formatting here: https://pandas.pydata.org/pandas-docs/stable/user_guide/style.html#Table-styles, by default None
- **column_order** (*List[str], optional*) – Specify the columns and ordering in the final visualization. Overrides version_info, so you must specify version columns in addition. Defaults to None, which will set the column order to a preset useful for VtR.

get_visualization() :

returns a displayable visualizaton, can be displayed by calling display() in an interactive Python environment

get_visualization()

Returns a displayable object which can be displayed by calling display() in an interactive Python environment.

class ftpvl.visualizers.**SingleTableVisualizer**(*evaluation: ftpvl.evaluation.Evaluation*, *style_eval: ftpvl.evaluation.Evaluation*, *version_info: bool = False*, *custom_styles: List[dict] = None*, *column_order: List[str] = None*)

Represents a visualizer for a styled single table, possibly with version information.

Parameters

- **evaluation** (Evaluation) – the Evaluation with values to display
- **style_eval** (Evaluation) – the Evaluation to use for styling. Should be processed using a Style, all values are valid CSS strings or empty.
- **version_info** (*bool, optional*) – Flag to display version information from the build results in the final visualization, by default False

- **custom_styles** (*List[dict], optional*) – Specify additional styling for the final visualzation. See formatting here: https://pandas.pydata.org/pandas-docs/stable/user_guide/style.html#Table-styles, by default None
- **column_order** (*List[str], optional*) – Specify the columns and ordering in the final visualization. Overrides version_info, so you must specify version columns in addition. Defaults to None, which will set the column order to a preset useful for VtR.

get_visualization() :
returns a displayable visualizaton, can be displayed by calling display() in an interactive Python environment

get_visualization()
Returns a displayable object which can be displayed by calling display() in an interactive Python environment.

3.1.6 Enums

Direction

class ftpvl.processors.**Direction**
Represents the optimization direction for certain test metrics. For example, runtime is usually minimized, while frequency is maximized.

MAXIMIZE = 1
Indicates that the corresponding metric is optimized by *maximizing* the value

MINIMIZE = -1
Indicates that the corresponding metric is optimized by *minimizing* the value

Core API Learn about the API of FTPVL.

Python Module Index

f

Index