Rules for Writing Metrics

Predicate metrics are a Python class with a defined structure. This class does not contain specific dependencies other than `pandas`.

Code examples of metrics are available on a separate page.

This page describes:

1. The structure of the metric class
2. Recommendations for writing metrics

Structure of the Metric Class

The metric class has a set of mandatory fields and methods. There are no restrictions on using additional fields or methods. We even recommend using additional methods and fields in the "Recommendations for Writing Metrics" section.

Mandatory fields:

1. `__desc__`;
2. `__tags__`;
3. `is_scalar`;
4. `is_signal`;

Mandatory methods:

1. `__init__`;
2. `__call__`;
3. `save`;
4. `scalar` (if the flag `is_scalar = True`);
5. `signal` (if the flag `is_signal = True`).

The sections below describe the fields and methods in more detail.

Fields

The class name becomes the name of the metric in the system.

`__desc__` - a human-readable description of the metric.

`__tags__` - a list of tags assigned to the metric in the system.

`is_scalar` - a flag indicating that the metric has a single scalar result. If `is_scalar = True`, the metric must contain the `scalar` method.

`is_signal` - a flag indicating that the metric has a signal. If `is_signal = True`, the metric must contain the `signal` method.

Thus, the metric class "Kolmogorov-Smirnov Test," which contains both a graph and a scalar value, as well as a signal, should start as follows:

```
class r_2_5_KS_on_scale:
    __desc__ = "KS-test on scale. Kolmogorov-Smirnov Test"
    __tags__ = ["risk", "scalar"]

    is_scalar = True
    is_signal = True
```

Methods

__init__

The class initialization method is designed for inputting metric parameters. All metric parameters will be passed to this method when calling the metric in the Predicate project or within the library.

Basic signature:

```
def __init__(self, *, **kwargs: typing.Any) -> None:
    ...
```

The definition of the `__init__` method in the metric class must comply with the following rules:

1. At least one dataset must be passed to the method.
2. Each parameter must have a type annotation according to the metric parameterization rules.
3. The default parameter value must not violate the logic of validation of the specified parameter type.
4. Parameter values must be passed to subsequent methods through the instance attributes `self`.

>>> **Parameter Types of Metrics and Their Usage** <<<

We recommend performing all checks of the entered values within this method.

Example for the "Kolmogorov-Smirnov Test" metric:

```
def __init__(
        self,
        df: pd.DataFrame,
        scale_column: str,
        target_column: str,
        threshold_yellow: float = 10,
        threshold_red: float = 30,
    ):
        self.scale_column = scale_column
        self.target_column = target_column
        self.df = df.astype({self.target_column: "float"})
        self.threshold_yellow = threshold_yellow
        self.threshold_red = threshold_red

        if self.df.empty:
            raise Exception("Dataframe is empty")
        if self.target_column not in self.df:
            raise ValueError(f"Field {self.target_column} does not exist in
the dataframe")
        if self.scale_column not in self.df:
            raise ValueError(f"Field {self.scale_column} does not exist in the
dataframe")
        if self.df[self.scale_column].nunique() > 100:
            raise Exception("Error: scale variable is not categorical")
```

Thus, the "Kolmogorov-Smirnov Test" accepts:

- A `pandas` dataframe with data for calculation;
- The name of the column with the scale;
- The name of the column with the target variable;
- Yellow and red thresholds for the signals.

In this example, the signal thresholds have default values.

In the body of the method, the metric instance (`self`) is assigned parameter values for use in other methods, and checks for the correctness of the entered data are performed.

> ***__call__***

The class call method contains the main calculations of the metric. Usually, it includes the block for composing the metric graphs, but there are no restrictions on the content; the method can contain any calculations.

Basic signature:

```
def __call__(self) -> None:
    ...
```

The method has no parameters other than `self` and does not return anything. All calculated values must be saved in the instance attributes `self`.

Example for the "Kolmogorov-Smirnov Test" metric:

```
def __call__(self) -> None:
        dataset = self.df.loc[:, [self.target_column,
self.scale_column]].dropna()

        # the rank number of the rating scale (the method of obtaining depends
on the format of the data in the column self.scale)
        dataset["bin_number"] = dataset[self.scale_column].map(
            lambda x: int(x.split("_")[-1])
        )  # dataset[self.scale].astype('category').cat.codes#

        dataset = dataset.sort_values(by=["bin_number"], ascending=False)

        good_cnt = dataset[dataset[self.target_column] == 0].shape[0]
        bad_cnt = dataset[dataset[self.target_column] == 1].shape[0]

        gr_bad = (
            pd.DataFrame(
                dataset.groupby("bin_number", observed=False)
[self.target_column].sum()
            ).cumsum()
            / bad_cnt
        )
        dataset["target_inverse"] = np.where(dataset[self.target_column] == 1,
0, 1)
        gr_good = (
            pd.DataFrame(
                dataset.groupby("bin_number", observed=False)
["target_inverse"].sum()
            ).cumsum()
            / good_cnt
        )

        ks_calc_temp = pd.merge(gr_good, gr_bad, how="left", left_index=True,
right_index=True)
        ks_calc_temp["diff"] = 0
        ks_calc_temp["diff"] = abs(
            ks_calc_temp.iloc[:, 0:1].values - ks_calc_temp.iloc[:,
1:2].values
        )

        self.scalar_value = 100 * ks_calc_temp["diff"].max()
        ks_result = round(self.scalar_value, 2)
        result_idx = ks_calc_temp["diff"].argmax()

        x_value1 = gr_bad.index.values.tolist()
        y_value1 = gr_bad[self.target_column].values.astype("float").tolist()
        x_value2 = gr_good.index.values.tolist()
```

```
        y_value2 = gr_good["target_inverse"].values.astype("float").tolist()
        x_value3 = [
            float(gr_bad.index.values[result_idx]),
            float(gr_bad.index.values[result_idx]),
        ]
        y_value3 = [
            float(gr_bad[self.target_column].values[result_idx]),
            float(gr_good["target_inverse"].values[result_idx]),
        ]

        line1 = go.Scatter(
            mode="lines",
            x=x_value1,
            y=y_value1,
            name="bad",
            line={"width": 3},
            marker={"color": "#63666A"},
        )
        line2 = go.Scatter(
            mode="lines",
            x=x_value2,
            y=y_value2,
            name="good",
            line={"width": 3},
            marker={"color": "#3eb489"},
        )
        line3 = go.Scatter(
            mode="lines",
            x=x_value3,
            y=y_value3,
            name=f"KS-statistic = {ks_result.astype('float')}",
            marker={"color": "black"},
        )
        self.fig = go.Figure(data=[line1, line2, line3])
        self.fig.layout = self.custom_layout()
```

Here, for the "Kolmogorov-Smirnov Test" metric, the following parameters were calculated:

- `self.scalar_value` - the scalar value for this metric;
- `self.fig` - the Plotly graph of the metric.

If the metric returns only scalar values, it is permissible to skip the description of this method. For example, for the "Maximum Value" metric:

```
def __call__(self) -> None:
        pass
```

> ***save***

The `save` method is intended for saving all graphs to files for further output in reports.

Basic signature:

```
def save(self, output_dir: str) -> dict[str, str] | None:
    ...
```

The method takes the path to the save folder as a string variable `output_dir`. The method returns a dictionary of values : .

Predicate can only process data in the following formats:

1. HTML;
2. PNG;
3. JPG;
4. SVG.

You **can** save the metric results in another format, but you need to create an HTML file in this same method and embed a link to your file in an unknown format for Predicate. This way, the system will be able to display it in the report.

Example of the `save` method for the "Kolmogorov-Smirnov Test" metric:

```
def save(self, output_dir: str) -> dict[str, str] | None:
        self.fig.write_html(
            f"{output_dir}/data.html",
            config={"displaylogo": False},  # remove the plotly logo
        )
        return {f"scale_{self.scale_column}": f"{output_dir}/data.html"}
```

In this case, one graph from the attribute `self.fig` was saved. However, it is also possible to save multiple graphs or save one or multiple graphs conditionally, as in the "Density Distribution" metric:

```
def save(self, output_dir: str) -> dict[str, str] | None:
        if self.split_charts:
            result = {}
            for column_name, fig in self.figs.items():
                file_path = f"{output_dir}/data_{column_name}.html"
                fig.write_html(
                    file_path,
                    config={"displaylogo": False},  # remove the plotly logo
                )

                result[column_name] = file_path
            return result
        else:
            self.fig.write_html(
                f"{output_dir}/data.html",
                config={"displaylogo": False},  # remove the plotly logo
            )

            return {"fig_name": f"{output_dir}/data.html"}
```

If the metric does not return any graphs, it is permissible to skip the description of this method. For example, for the "Maximum Value" metric:

```
def save(self, output_dir: str) -> dict[str, str] | None:
        pass
```

scalar

This method should appear in the metric class if the flag `is_scalar = True`.

The method is intended for calculating the numerical value of the metric.

Basic signature:

```
def scalar(self) -> int | float:
    ...
```

The method does not take any parameters. The method returns a numerical value as `int` or `float`.

For non-scalar metrics (flag `is_scalar = False`), this method does not need to be defined. If defined, Predicate will perceive it as an error.

Example of the `scalar` method for the "Kolmogorov-Smirnov Test":

```
def scalar(self) -> int | float:
        return self.scalar_value
```

In this case, the scalar value was calculated in advance, and it only needed to be returned from the method.

For metrics without graphs, it is typical to include the logic for calculating the numerical value in the `scalar` method. For example, for the "Maximum Value" metric:

```
def scalar(self) -> int | float:
        self.scalar_value = float(self.df[self.field_column].min(skipna=True))

        return self.scalar_value
```

signal

This method should appear in the metric class if the flag `is_signal = True`.

The method is intended for calculating the metric's signal.

The signal is an indicator of the quality of the metric. It is expressed in three colors:

- Green (`green`) - metric values are within the norm.

- Yellow (`yellow`) - metric values raise concerns.
- Red (`red`) - metric values are far beyond the norm.

Basic signature:

```
def signal(self) -> Literal["red", "yellow", "green"]:
    ...
```

The method does not take any parameters. The method returns a string value of the signal. The signal values must strictly be from the list `["red", "yellow", "green"]`.

Example of the `signal` method for the "Kolmogorov-Smirnov Test":

```
def signal(self) -> Literal["red", "yellow", "green"]:
        signal_light = "green"

        if self.scalar_value > self.threshold_red:
            signal_light = "red"
        elif self.scalar_value > self.threshold_yellow:
            signal_light = "yellow"

        return signal_light
```

Order of Method Execution

The methods in both the Predicate product and the Predicate library are executed in the following order:

1. `__init__`;
2. `__call__`;
3. `save`;
4. `scalar` (if the flag `is_scalar = True`);
5. `signal` (if the flag `is_signal = True`).

Consider the order of method execution when developing the metric. It affects how values can be passed from method to method.

Recommendations for Writing Metrics

In addition to explicit requirements for the structure of the metric class, there are several optional recommendations for writing them. These recommendations simplify working with the metric: its support by different team members; creating multiple metrics based on a similar template; checking the data entered into the metric.

The list of recommendations is not exhaustive. We will expand it as new ideas arise.

Method `custom_layout`

If your metric returns a graph as a **Plotly** object, we recommend creating a `custom_layout` method.

Layout is the layout of the graph. The layout describes the display parameters of the graph: title; axis titles; boundaries of displayed axes; and many other parameters. More about this can be found in the Plotly documentation.

The layout usually takes about ten lines of code. The `custom_layout` method allows you to avoid cluttering the `__call__` method with unnecessary descriptions and move them to a separate method.

The `custom_layout` method simplifies copying code from metric to metric. In your Plotly graph description, only the call to the `custom_layout` method will appear, so you can transfer this graph to another metric with a different layout without additional code rewriting.

Example of the `custom_layout` method for the "Kolmogorov-Smirnov Test":

```
def custom_layout(self) -> Optional[Dict[str, Any]]:
        return {
            "title": {"text": "<b>Kolmogorov-Smirnov Test</b>", "x": 0.1, "y":
0.97},
            "legend": {"yanchor": "bottom", "y": 0.05, "xanchor": "right",
"x": 1},
            "yaxis": {"title": "Cumulative Share", "side": "left"},
            "xaxis": {
                "title": "Rank Number of the Rating Scale",
                "side": "left",
                "type": "category",
                "domain": [0, 0.8],
            },
            "margin": {"t": 35, "b": 5, "l": 5, "r": 5},
        }
```

Field `scalar_value`

We recommend assigning the numerical value of the metric to the `scalar_value` attribute. This field simplifies life: it becomes easier to read the code; it is always clear what to return in the `scalar` method; it is easy to refer to the numerical value in other methods.