

Adding a Metric

The metrics are implemented in a way so that adding a new metric is straightforward.

All of the abstract metric classes are defined in `syntherela.metrics.base`. To add a new metric, you need to create a new class that inherits from one of the abstract classes. All of our metrics inherit the `BaseMetric` class from the SDV package. We divide the metrics into 3 categories based on the type: `Statistical`, `Distance`, and `Detection`, and into 3 categories based on the granularity: `SingleColumnMetric`, `SingleTable`, and `MultiTable`.

BaseMetric Class

All of our metrics inherit the `BaseMetric` class from the SDV package. The `BaseMetric` class has a `compute` function, as well as the class variables `name`, `goal` (maximize or minimize), `min_value` and `max_value`. All of the metrics should implement the `compute` function, which calculates the metric value and returns a dictionary with the results of the metric. The `goal` variable should be set to `maximize` or `minimize` based on the metric. The `min_value` and `max_value` variables should be set to the minimum and maximum possible values of the metric.

In our abstract metric classes we implement a `run` function additionally to the `compute` function. When the benchmark is run, the `run` function is called, which usually validates the data, calls the `compute` function and validates the results.

Statistical metrics

Statistical metrics calculate the metric value and the p-value. The `StatisticalBaseMetric` class is written in a way that when adding a new metric you only need to implement the `compute` function and the `validate` function. The `compute` function accepts an original and synthetic col returns a dictionary of the form:

```
{"statistic": <statistic>, "p_value": <pval>}
```

The `validate` function should be implemented in the metric class and should return a boolean value indicating whether the metric results are valid. The `validate` function is called before the `compute` function on the real and synthetic data. If the `validate` function returns `False`, the metric is skipped for that specific column, table or dataset it is computed on.

Distance metrics

Distance metrics calculate the distance between the original and synthetic data. The `DistanceBaseMetric` class is written in a way that when adding a new metric you only need to implement the `compute` function. The `compute` function accepts an original and synthetic col, table or dataset and returns the metric value (distance). Then, the metric automatically calculates the reference mean, variance and confidence interval by bootstrapping the original data, which is used to decide whether the metric has detected that the data is synthetic or not. Then, the metric also calculates the bootstrap mean and standard error of the metric by bootstrapping the real and synthetic data. Finally, the metric returns a dictionary of the form:

```
{'value': value,
 'reference_mean': reference_mean,
 'reference_variance' : reference_variance,
 'reference_ci': reference_standard_ci,
 'bootstrap_mean': bootstrap_mean,
 'bootstrap_se': bootstrap_se}
```

Detection metrics

Detection metrics alongside the real and synthetic data also accept a scikit-learn API based classifier to calculate the accuracy of the classifier on the real and synthetic data. The `DetectionBaseMetric` class is written in a way that when adding a new metric you don't need to implement anything, or you can overwrite any of the functions. The `run` function calculates the accuracy and standard error of the classifier on the real and synthetic data and then the p values of the binomial test for the detection of synthetic data or detection of potential data copying. The function returns a dictionary of the form:

```
{"accuracy": np.mean(scores),
 "SE": standard_error,
 "bin_test_p_val" : np.round(bin_test_p_val, decimals=16),
 "copying_p_val": np.round(copying_p_val, decimals=16)}
```

The metric also implements a `prepare_data` function where the data gets transformed from a pandas dataframe to a numpy array. The `prepare_data` function can be overwritten if the data needs to be transformed in a different way, it just has to return the `X` and `y` arrays. If the classifier supports feature importance there is also a function `plot_feature_importance`.

Single Column and SingleTable Metrics

For granularity, a `SingleColumnMetric` and `SingleTableMetric` classes are provided where a function `is_applicable` should be implemented for the single column metrics. The function accepts a column type as a string (for single column) or metadata (for single table) and should return a boolean whether the metric is applicable to the column type/table. For single table metrics the `is_applicable` function is already implemented and checks whether the table contains at least one column which is not a primary or foreign key.

Example

Here is an example of how to add a new metric to the benchmark. Let's say we want to add a new distance metric that calculates absolute difference of the means of the real vs synthetic column. We would create a new class that inherits from `DistanceBaseMetric` and `SingleColumnMetric`, and implement the `compute` and `is_applicable` functions.

```
class AbsoluteMeanDistance(DistanceBaseMetric, SingleColumnMetric):

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.name = "AbsoluteMeanDistance"
        self.goal = Goal.MINIMIZE
        self.min_value = 0.0
        self.max_value = float('inf')

    @staticmethod
    def is_applicable(column_type):
        return column_type in ["numerical"]

    @staticmethod
    def compute(real_data, synthetic_data, **kwargs):
        orig_col = pd.Series(real_data).dropna()
        synth_col = pd.Series(synthetic_data).dropna()

        return np.abs(orig_col.mean() - synth_col.mean())
```

For more examples look at the implementations of the metrics in `syntherela.metrics`.

