A Grammar of Data Analysis

Xunmo Yang, Taylor Pospisil, Omkar Muralidharan, Google, Inc.
and
Dennis L. Sun
Google, Inc. and Stanford University

October 3, 2025

Abstract

This paper outlines a grammar of data analysis, as distinct from grammars of data manipulation. The primitives of this grammar are metrics and dimensions. We describe a Python implementation of this grammar called Meterstick, which is agnostic to the underlying data source, which may be a DataFrame or a SQL database.

Keywords: data science, grammar of data analysis

1 Introduction

This paper proposes a grammar of data analysis, in the vein of grammars that have been proposed for graphics (Wilkinson 2006, Wickham 2010). Just as a grammar of language specifies the primitives that make up a sentence (e.g., subject, verb, predicate), a grammar of data analysis specifies the primitives that make up an analysis (e.g., metrics, dimensions). By identifying these primitives, as well as the rules for composing them, we can design software libraries where those primitives are promoted to first-class citizens. The goal is to streamline the data analysis workflow, much as `ggplot2` has done for data visualization. (Wickham 2009)

We are not the first to propose such an idea. Many people regard `dplyr` (Wickham & Francois 2014) as a kind of grammar for data analysis. While `dplyr` is indeed a grammar, its primitives are too low a level to facilitate the more complex analyses we have in mind, as we argue in Section 3. In fairness to `dplyr`, it only claims to be a "grammar of *data manipulation*," which is a precursor to data analysis. In order to facilitate complex analyses, a good grammar of data analysis ought to provide a layer of abstraction above the details of data manipulation. That said, implementations of a grammar of data analysis can and should be built on top of a good grammar of data manipulation, such as `dplyr` in R or `pandas` in Python.

2 A Grammar of Data Analysis

Many data analysis problems can be framed as calculating **metrics** over **dimensions**. A metric defines the computation, while a a dimension defines the granularity at which that computation should be done. For example:

1. A baseball team wants to track the distribution of pitch types thrown by each pitcher to each batter. The "dimensions" are the pitcher and the batter, and the "metric" is a vector of probabilities representing the distribution over the different pitch types.

2. After running randomized experiments to evaluate 5 proposed new website designs, an online retailer wants to know the change in churn rate in each region from each

of the new designs. The "dimension" is the region, and the "metric" is a vector of changes in churn rate between each of the new designs and the control design.

3. Card & Krueger (1994) estimated the effect of minimum wage on employment using a natural experiment where the minimum wage increased in New Jersey but stayed the same in Pennsylvania. They measured employment at different fast-food restaurants before and after the minimum wage increase. Here, the "dimension" is the restaurant, and the "metric" is the difference-in-difference (DID) in the employment rate.

At a high level, a grammar of data analysis consists of just these two components, metrics and dimensions, which can be specified independently of one another. That is, we can easily imagine calculating the same metric over a different set of dimensions (e.g., pitcher only, instead of both pitcher and batter), or calculating a different metric (e.g., batting average) over the same set of dimensions. We illustrate this grammar using our implementation of it, an open-source Python library called Meterstick:

1. In the baseball example, the metric is simply a count over pitch types, and the Meterstick code reflects this:

```
pitch_types = Count("pitchtype") | Distribution(over="pitchtype")
pitch_types.compute_on(df, split_by=["pitcher", "batter"])
```

2. In the online retail example, we transform the basic churn metric by the `PercentChange` operation to obtain the final metric:

```
churn = (Sum("lost") / Count("lost")).set_names(["churn"])
churn_change = churn | PercentChange("experiment", "control")
churn_change.compute_on(df, split_by=["region"]))
```

3. In the minimum wage example, we can implement the DID estimator for the average employment rate as

```
employment_rate = Mean("EMP")  # equals Sum("EMP") / Count("EMP")
did = (employment_rate
       | AbsoluteChange("STATE_NAME", "PA")
       | AbsoluteChange("PERIOD", "Before"))
did.compute_on(df)
```

Although we calculated the absolute change in employment to reproduce the results of Card & Krueger (1994), we could just have well calculated the percent change in employment, instead reporting a percentage point difference. This is not straightforward to do in the regression framework used by Card & Krueger (1994), but it is straightforward in this grammar of data analysis; we would simply need to replace the first `AbsoluteChange` by a `PercentChange`.

In each case, a metric is defined and piped to the function `compute_on()` that finally computes that metric on data. The dimensions are specified in the `split_by=` argument of `compute_on()`. If we decide to change the dimensions later, the code only needs to be modified in one place, making the code more modular.

Compared to dimensions, metrics are much richer in their possibilities. The remainder of this section demonstrates how the algebraic structure of metrics can be used to construct arbitrarily complex metrics.

2.1 The Algebra of Metrics

Formally, a **metric** is a function f that maps the space of dataframes $\mathcal{D}$ to a space of arrays, which may be one of:

- the space of scalars $\mathbb{R}$ (e.g., sum, mean, count),
- the space of vectors $\mathbb{R}^n$ (e.g., distribution), or
- the space of matrices $\mathbb{R}^{m \times n}$.

We can add, subtract, multiply, and divide metrics f_1 and f_2 according to the general rule

$$(f_1 \circ f_2)(D) = f_1(D) \circ f_2(D),$$

assuming that the operation $\circ$ is already well-defined on arrays. For example, the churn metric we defined above is really a function of a dataframe D that is computed as

$$\texttt{churn}(D) = \texttt{Sum("lost")}(D)/\texttt{Count("lost")}(D).$$

If the metric returns an array, then operations are typically performed element-wise, although operations can be "broadcast" for arrays of different shapes. (Van Der Walt et al. 2011) If f_1 and f_2 may produce arrays of incompatible shapes, then $(f_1 \circ f_2)(D)$ is NaN.

The space of metrics $\mathcal{M}$ resembles an algebraic field. (Dummit & Foote 2004) It is helpful to analogize the space of metrics $\mathcal{M}$ to a field because then we can regard adding a new metric as a field extension. Just as extending the field of rationals $\mathbb{Q}$ to include $\sqrt{2}$ introduces infinitely many new elements of the form

$$\{a + b\sqrt{2} : a, b \in \mathbb{Q}\},$$

defining a new metric—say, `SD("x")`—introduces infinitely many new metrics of the form

$$\{m_1 + m_2 \cdot \texttt{SD("x")} : m_1, m_2 \in \mathcal{M}\}.$$

In particular, if we take $m_1 =$ `Mean("x")` and $m_2 =$ `1.96 / Count("x") ** 0.5`, then the new metric is familiar as the upper bound of a 95% confidence interval.

2.2 Operations

Metrics can also be modified by `Operation`s. All of our above examples use `Operation`s:

1. In the baseball example, the `Count` metric is modified by `Distribution(over="pitchtype")`. This operation has two effects: (1) it returns a value for each pitch type instead of a single aggregate value, and (2) it normalizes these values so that they sum to 1.

2. In the online retail example, the `churn` metric (itself a composite of two metrics) is modified by `PercentChange("experiment", "control")`. This operation similarly alters `churn` in two ways: (1) it computes the churn rate separately for each experiment arm, and (2) it reports the percent change in churn relative to the control arm.

3. In the minimum wage example, the `employment` metric is first modified by `AbsoluteChange("state", "PA")`, and subsequently by `AbsoluteChange("period", "before")`. Each operation has an effect analogous to `PercentChange` in the previous example. The

net effect is that we: (1) compute the average employment rate for each state-period combination, (2) then calculate the difference between states (PA − NJ) for both the period before and after the minimum wage increase, and (3) calculate the difference between periods (after − below) of those differences. The chaining of `Operations` is possible because an `Operation` is itself a `Metric` that can be modified by another `Operation`. Therefore, `Operations` can be chained indefinitely.

One important class of operations are "standard error" operations, which specify a method of calculating standard errors. For example, the following code would produce bootstrap standard errors for the difference-in-differences metric above:

```
(employment_rate
 | AbsoluteChange("STATE_NAME", "PA")
 | AbsoluteChange("PERIOD", "Before"))
 | Bootstrap()
).compute_on(df)
```

Since we often need to quantify uncertainty in metrics, "standard error" operations are very commonly chained with other operations.

Continuing the analogy to field extensions, we can think of adding an operation as analogous to adding the $\sqrt{\cdot}$ operator to $\mathbb{Q}$. This introduces not only $\sqrt{2}$, but also $\sqrt{3}$ and $\sqrt{-1}$ to the field. That is, adding an operation is like a field extension of infinite degree. Operations vastly expand the space of possible metrics, far beyond what we could hope to achieve by adding new metrics individually.

Operations must modify other metrics; they cannot stand alone as metrics. Just as the $\sqrt{\cdot}$ operator is not a number, the percent change is not a viable metric by itself. We must specify a metric of which to calculate the percent change.

By composing metrics and applying operations, we can construct arbitrarily complex metrics. We can also combine these metrics with different dimensions. To complete the data analysis, we finally need to specify a data source. These three components of a data analysis are diagrammed in Figure 1. The input and output data for the churn rate computation, without any dimension, are illustrated in Figure 2.

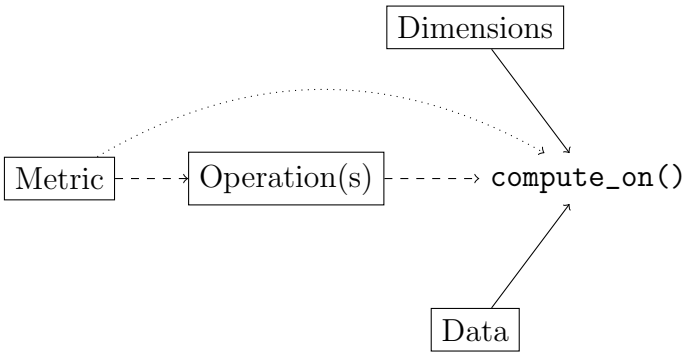

Figure 1: The basic flow of a data analysis.

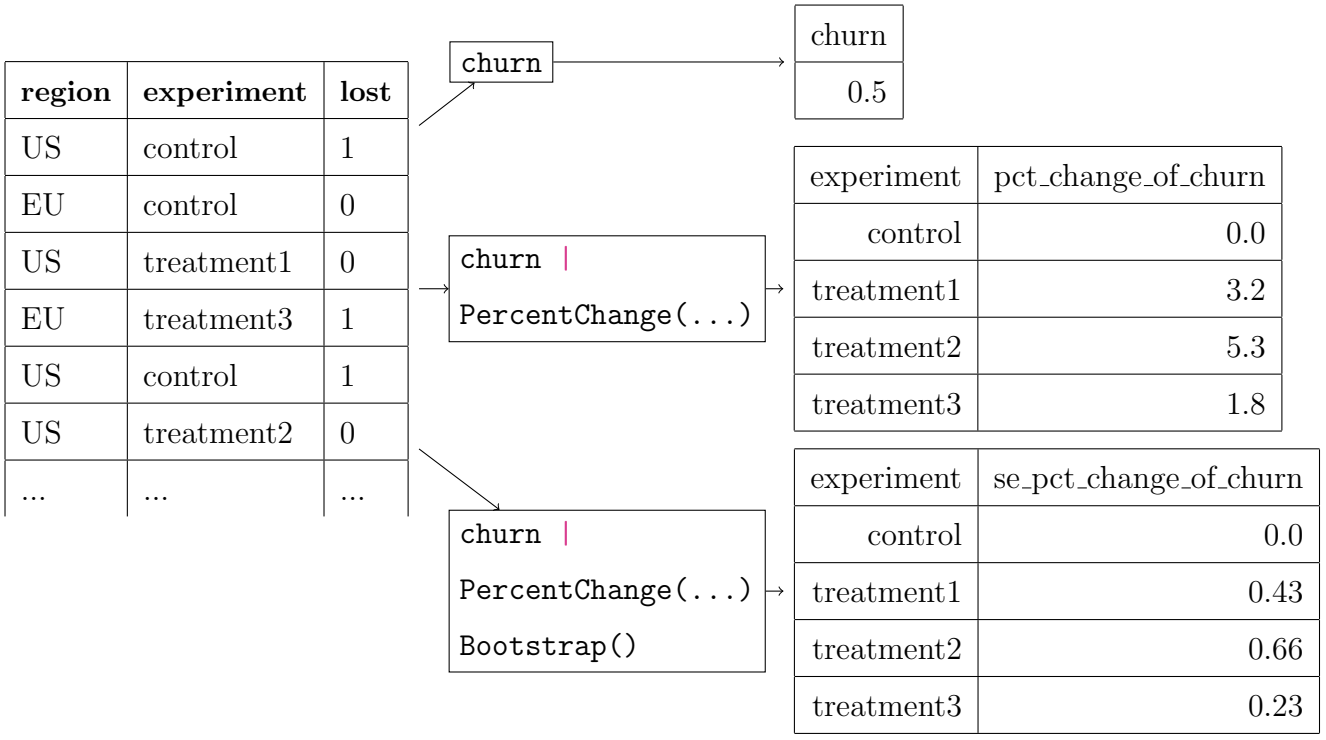

Figure 2: The input and output data schema for the churn rate computation.

3 Comparison with Grammars of Data Manipulation

The idea of calculating metrics over dimensions is reminiscent of the "split-apply-combine" framework of Wickham (2011). We first *split* the data into many smaller data sets, one for each combination of dimensions, called **slices**. Then, we *apply* the metric to each slice. Finally, we *combine* the results in the output.

In many packages (Gray et al. 1997, McKinney et al. 2010, Wickham & Francois 2014), the "split-apply-combine" framework is implemented using the "group-by" operator. The user specifies variables to group by and an aggregation function. However, the grouping variables do not always correspond neatly to the dimensions, especially when metrics are modified by operations. To see why, consider the examples from above:

1. To implement the baseball example in `dplyr` in `R` (the `pandas` code would be similar), the grouping variables have to include the dimensions *and* the pitch types:

   ```
   df %>% group_by(pitcher, batter, pitchtype) %>%
          count() %>%
          group_by(pitcher, batter) %>%
          mutate(n / sum(n))
   ```

   The analysis requires two passes through the data. In the first pass, we count at-bats, grouping by the dimensions *and* the pitch type. Then, we aggregate again, but this time grouping only by the dimensions, in order to calculate the distribution over the pitch types.

   The problem with this workflow is that it is fragile. Suppose we wanted to add a dimension to this analysis (e.g., `month`). We would have to change the code above in multiple places (i.e., both `group_by`s) to effect this one change. Contrast this with the Meterstick code above, where the dimensions are specified in only one place.

2. To implement the online retailer example in `dplyr`, we again have to make two passes through the data.

   ```
   df_by_expt <- df %>% group_by(region, experiment) %>%
   ```

```
                        summarize(churn=sum(lost) / n())
df_treated <- filter(df_by_expt, experiment != "control")
df_control <- filter(df_by_expt, experiment == "control")
df_treated %>% inner_join(df_control,
                          split_by="region",
                          suffix=c("_treated", "_control")) %>%
               mutate(churn_diff=100 * (churn_treated / churn_control - 1))
```

In the first pass, we group by the dimensions *and* the experiment. In the second pass, we join the treated data to the control along the dimensions and calculate the difference. (This generalizes to more than one treatment arm.) Again, if we wanted to add a dimension to this analysis, we would need to modify the code in multiple places: `group_by()` and `inner_join()`.

3. To implement the difference-in-differences analysis in `dplyr`,

```
df %>%
  group_by(STATE_NAME, PERIOD) %>%
  summarize(EMP_RATE = mean(EMP, na.rm=T)) %>%
  pivot_wider(names_from = STATE_NAME,
              values_from = EMP_RATE) %>%
  mutate(DIFF=NJ - PA) %>%
  select(PERIOD, DIFF) %>%
  pivot_wider(names_from = PERIOD,
              values_from = DIFF) %>%
  mutate(DIFF_OF_DIFFS=After - Before)
```

If we also wanted to report standard errors for these metrics, then the code would be still more complex. The problem is that the syntax does not mirror intention; a single element of the analysis corresponds to several locations in the code. The above examples highlight the "bottom-up" nature of data manipulation, requiring the analyst to specify

low-level details such as what variables to group by at each stage. By contrast, a grammar of data analysis should be "top-down," with its primitives matching the elements of the analysis.

On a practical level, the bottom-up code presented in this section is ill-suited to the iterative process of data science, where questions are constantly being tweaked and refined. Data science demands code that is reproducible, reusable, and reviewable. We have already seen how the bottom-up code is not modular, where changing one element of the analysis requires changing multiple lines of code.

4 Implementation

In Section 2, we described the interface for Meterstick, a particular implementation of the grammar of data analysis. In this section, we discuss implementation details that further illustrate the recursive and composable nature of this grammar.

Before we dive into the details, we emphasize that this implementation is separate from the grammar; the analyst does not need to understand these implementation details. Once they specify their intention in the grammar, the library handles the details of carrying out the analysis, whether on a DataFrame or a SQL database. We specify the implementation details here because we think they are of independent interest.

The key idea behind the implementation is that every `Metric`, no matter how complex, can be represented as a tree of `Metric`(s). For example, the change in the churn rate with bootstrap standard errors is represented internally by the tree shown in Figure 3. An `Operation` is simply a `Metric` with children. Each `Metric` has methods which calculate the metric on data.

4.1 DataFrames

When data is in a DataFrame, the metric is evaluated using the `compute_on` method. This method specifies how to calculate the metric on a given DataFrame over a given set of dimensions. The implementation differs depending on whether the `Metric` is an `Operation` or not.

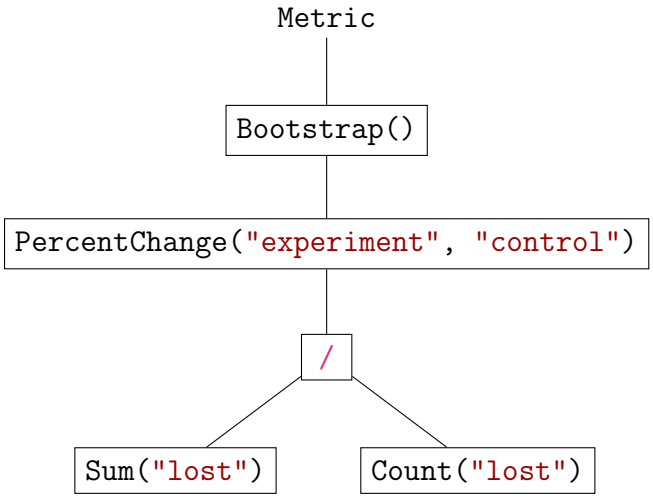

Figure 3: Example structure of the metric from the online retailer example.

For simple `Metrics` which are not `Operations`, such as `Sum`, the computation can be accomplished by simply *grouping by* the dimensions. Each type of `Metric` simply needs to implement the `compute()` method and specify a default naming pattern, as shown below.

```
class Metric:

  def compute_on(self, data, split_by=None):
    if split_by:
      res = self.compute(data.groupby(split_by))
    else:
      res = [self.compute(data)]
    res = pd.DataFrame(res)
    res.columns = self.names
    return res

  def set_names(self, names):
    self._names = names
    return self

  @property
  def names(self):
    return getattr(self, '_names', self.default_names)
```

```
@attrs.define
class Sum(Metric):
  var: str

  def compute(self, data):
    return data[self.var].sum()

  @property
  def default_names(self):
    return [f'sum_{self.var}']
```

For `Operation`s, the computation requires passing data *down* and results *up* the tree. Conceptually, the computation can be organized into three steps:

1. preprocess the data,
2. call the `compute_on` methods of the children on the preprocessed data, and
3. process the results from the children to compute the final metric.

The general `Operation` class looks as follows.

```
class Operation(Metric):

  def compute_on(self, data, split_by=None):
    data_preprocessed = self.preprocess(data, by)
    child_res = self.compute_children(data_preprocessed, by)
    return self.process_results(child_res, by)
```

Each `Operation` simply needs to implement the three methods above. Table 1 contains implementations for the `Operation`s shown in Figure 3.

4.2 SQL

Analyzing data using DataFrames becomes infeasible when the dataset size exceeds available computer memory. In these situations, data is typically stored in relational databases

and analyzed with SQL queries. Therefore, each `Metric` also has a `to_sql` method that generates a SQL query which calculates the metric.[1]

As before, the implementation differs according to whether the `Metric` is an `Operation` or not. For simple `Metric`s, the computation can be carried out in a single query. Each type of `Metric` needs only to specify a SQL expression that computes it, as shown below.

```
class Metric:

  def sql_aggregate(self, data, dimensions):
    # Helper function for constructing aggregation queries.
    dim_sql = ','.join(dimensions) + ',' if dimensions else ''
    groupby = f'GROUP BY ' + ','.join(dimensions) if dim_sql else ''
    val_cols = ','.join([f'{s} AS {n}'
                         for s, n in zip(self.sql, self.names)])
    return f'SELECT {dim_sql} {val_cols} FROM {data} {groupby}'

  def to_sql(self, data, split_by=None):
    return self.sql_aggregate(data, split_by)


@attrs.define
class Sum(Metric):
  var: str

  @property
  def sql(self):
    return [f'SUM({self.var})']
```

For `Operation`s, we organize SQL generation into the same three steps (preprocess, compute on children, and process results) as for DataFrames, except that each step now returns a SQL subquery instead of a DataFrame. Because each child may add new dimensions to the data, we keep track of these dimensions in the property `extra_dims`. (This was not necessary for DataFrames because these dimensions could be stored directly in the index of the DataFrame itself.)

[1]The SQL dialect in the following implementation is GoogleSQL.

```
class Operation(Metric):

  def sql_select(self, data, dimensions):
    # Helper function for constructing select queries.
    dim_sql = ','.join(dimensions) + ',' if dimensions else ''
    val_cols = ','.join([f'{s} AS {n}'
                         for s, n in zip(self.sql, self.names)])
    return f'SELECT {dim_sql} {val_cols} FROM {data}'

  def to_sql(self, data, split_by=None):
    data_preprocessed = self.preprocess_sql(data, split_by)
    children_query = self.children_to_sql(data_preprocessed, split_by)
    return self.assemble_query(children_query, split_by)

  @property
  def extra_dims(self):
    return []
```

Each `Operation` simply needs to implement the three methods in `to_sql`, as well as the `extra_dims` property. Table 2 contains implementations for the `Operations` in Figure 3.

4.3 Summary of Features

We have already seen how a grammar of data analysis provides flexibility and adaptability as needs evolve, and simplifies code review and maintenance. The Meterstick implementation of this grammar offers many built-in `Metrics`, such as `Sum`, `Mean`, `Quantile`, and `Variance`, and `Operations`, such as `Distribution`, `AbsoluteChange`, `Jackknife` and `Bootstrap`. It allows these operations to be chained indefinitely, as specified by the grammar. Moreover, Meterstick allows users to define their own metrics (by implementing the methods described above) and chain them with the built-in `Operations` to produce even more analysis pipelines. One common use case is that a data scientist develops a new metric and wants to calculate the metric, along with a confidence interval. If they implement the metric in Meterstick, then they get a bootstrap or jackknife confidence interval for free.

Meterstick is also able to abstract away the implementation details from the user. For

example, when calculating multiple metrics, some intermediate metrics may be shared across multiple metrics. In this situation, Meterstick caches the values of these intermediate metrics so that they can be reused, speeding up the computation.

Finally, Meterstick implements the grammar in a backend-agnostic way, meaning that the same `Metric` can calculated on a DataFrame or a SQL database by calling `compute_on` or `to_sql`, respectively. This allows analyses to be prototyped on smaller data sets and scaled seamlessly to data warehouses.

5 Conclusion

A grammar of data analysis must sit at a higher level of abstraction than a grammar of data manipulation, in order to represent common patterns of data analysis. In this paper, we have proposed a grammar that can flexibly describe any analysis that can be characterized as "calculating metrics over dimensions," in a way that allows frequent iteration over different metrics and different dimensions. We have seen how existing grammars of data manipulation fail to provide the necessary flexibility and modularity for these types of analyses. We have demonstrated how to implement this grammar in a backend-agnostic way so that it can be applied just as easily to small DataFrames in memory as to databases in the cloud.

This grammar of data analysis is built upon the idea that metrics can be composed and modified to produce more complex metrics. By providing a common language for expressing intention, it can improve communication and collaboration among analysts, promote best practices, and facilitate data exploration and metric development. Moreover, the grammar's focus on modularity and reusability can lead to more efficient and maintainable analysis pipelines.

SUPPLEMENTARY MATERIAL

Meterstick package An open-source implementation of this grammar of data analysis, as described in 4, is available at `https://www.github.com/google/meterstick`.

A Bootstrap Implementation

Below is the implementation of `resample_n_times` that is used in `Bootstrap`.

```
def resample_n_times(data, split_by, n_rep):
  by_sql = ','.join(split_by) + ',' if split_by else ''
  input_data = f'''
    SELECT
      *,
      ROW_NUMBER() OVER (PARTITION BY sample_idx) AS row_number,
      CEILING(RAND() * COUNT(*) OVER (PARTITION BY sample_idx))
        AS random_row_number,
    FROM {data},
    UNNEST(GENERATE_ARRAY(1, {n_rep})) AS sample_idx'''
  samples = f'''
    SELECT b.*
    FROM (
      SELECT
        {by_sql}
        sample_idx,
        random_row_number AS row_number
      FROM Data) AS a
    JOIN Data AS b
    USING ({by_sql} sample_idx, row_number)'''
  return (input_data, samples)
```

References

Card, D. & Krueger, A. B. (1994), 'Minimum wages and employment: A case study of the fast-food industry in new jersey and pennsylvania', *The American Economic Review* **84**(4), 772–793.

Dummit, D. S. & Foote, R. M. (2004), *Abstract algebra*, Wiley.

Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow,

F. & Pirahesh, H. (1997), 'Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals', *Data mining and knowledge discovery* **1**(1), 29–53.

McKinney, W. et al. (2010), Data structures for statistical computing in python, *in* 'Proceedings of the 9th Python in Science Conference', Vol. 445, Austin, TX, pp. 51–56.

Van Der Walt, S., Colbert, S. C. & Varoquaux, G. (2011), 'The numpy array: a structure for efficient numerical computation', *Computing in science & engineering* **13**(2), 22–30.

Wickham, H. (2009), *ggplot2: elegant graphics for data analysis*, Springer Science & Business Media.

Wickham, H. (2010), 'A layered grammar of graphics', *Journal of Computational and Graphical Statistics* **19**(1), 3–28.

Wickham, H. (2011), 'The split-apply-combine strategy for data analysis', *Journal of Statistical Software* **40**(1), 1–29.

Wickham, H. & Francois, R. (2014), 'dplyr: A grammar of data manipulation', *R package version 0.3. 0.2* .

Wilkinson, L. (2006), *The grammar of graphics*, Springer Science & Business Media.

Operation	`@attrs.define`
	`class Div(Operation):`
	`  child1: Metric`
	`  child2: Metric`
		`@attrs.define`
		`class PercentChange(Operation):`
		`  condition: str`
		`  baseline: str`
		`  child: Optional[Metric]=None`
			`@attrs.define`
			`class Bootstrap(Operation):`
			`  n_rep: int`
			`  child: Optional[Metric]=None`

`def preprocess(`
`  self,`
`  data,`
`  split_by):`
	`return data`	`return data`	`for i in range(self.n_rep):`
			`  yield data.sample(`
			`  frac=1, replace=True)`

`def compute_children(`
`  self,`
`  data,`
`  split_by):`
	`return (`
	`  self.child1.compute_on(`
	`  data, split_by),`
	`  self.child2.compute_on(`
	`  data, split_by))`
		`return self.child.compute_on(`
		`  data,`
		`  split_by + [`
		`  self.condition])`
			`sample_res = [`
			`  self.child.compute_on(`
			`  sample, split_by)`
			`  for sample in data]`
			`return pd.concat(`
			`  sample_res, axis=1)`

`def process_results(`
`  self,`
`  child_res,`
`  split_by):`
	`num, denom = child_res`
	`num.columns = self.names`
	`denom.columns = self.names`
	`return num / denom`
		`if split_by:`
		`  base = child_res.xs(`
		`  self.baseline,`
		`  level=self.condition)`
		`else:`
		`  base = child_res.loc[`
		`  self.baseline]`
		`res = child_res / base - 1`
		`res.columns = self.names`
		`return res * 100`
			`std = child_res.T.groupby(`
			`  level=0).std().T`
			`std.columns = self.names`
			`return std`

`@property`
`def default_names(self):`
	`return map('_div_'.join,`
	`  zip(self.child1.names,`
	`  self.child2.names))`
		`return [f'pct_change_of_{n}'`
		`  for n in self.child.names]`
			`return [f'se_{n}'`
			`  for n in self.child.names]`

Table 1: The computation for each `Operation` in Figure 3 requires three steps.

Operation	`@attrs.define`
	`class Div(Operation):`
	`  child1: Metric`
	`  child2: Metric`
		`@attrs.define`
		`class PercantChange(Operation):`
		`  condition: str`
		`  baseline: str`
		`  child: Optional[Metric]=None`
			`@attrs.define`
			`class Bootstrap(Operation):`
			`  n_rep: int`
			`  child: Optional[Metric]=None`

`def preprocess_to_sql(`
`  self, data,`
`  split_by):`
	`return data`	`return data`	`# resample_n_times is defined`
			`# in the Appendix due to space`
			`return resample_n_times(`
			`  data, split_by, self.n_rep)`

`def children_to_sql(`
`  self, data,`
`  split_by):`
	`return data`	`return self.child.to_sql(`
		`  data, split_by + [`
		`  self.condition])`
			`return (*data,`
			`  self.child.to_sql('Samples',`
			`  split_by + ['sample_idx']))`

`def assemble_query(`
`  self,`
`  child_res,`
`  split_by):`
	`return self.sql_aggregate(`
	`  child_res, split_by)`
		`dims = self.extra_dims + split_by`
		`u = ','.join(dims[1:])`
		`join = f'T JOIN Base USING ({u})'`
		`if not u:`
		`  join = 'T CROSS JOIN Base'`
		`return f'''`
		`WITH T AS ({child_res}),`
		`Base AS (SELECT *`
		`EXCEPT ({self.condition}) FROM T`
		`WHERE {self.condition}`
		`  = '{self.baseline}')`
		`{self.sql_select(join, dims)}'''`
			`(input_data, samples,`
			`  sample_res) = child_res`
			`sql = self.sql_aggregate(`
			`  'SampleRes',`
			`  by + self.extra_dims)`
			`return f'''`
			`  CREATE TEMP TABLE Data`
			`  AS ({input_data});`
			`  WITH Samples AS ({samples}),`
			`  SampleRes AS ({sample_res})`
			`  {sql}'''`

`@property`
`def sql(self):`
	`return map(' / '.join,`
	`  zip(self.child1.sql,`
	`  self.child2.sql))`
		`return [`
		`  f'(T.{c} / Base.{c} - 1) * 100'`
		`  for c in self.child.names]`
			`return [f'STDDEV({n})'`
			`  for n in self.child.names]`

`@property` `def extra_dims(self):`	`return []`	`return ([self.condition] +` `  self.child.extra_dims)`	`return self.child.extra_dims`

Table 2: The SQL generator for the `Operations` in Figure 3.