Mensor Documentation

Release v0.3.1

Matthew Wardrop

Apr 16, 2019

Contents

CHAPTER 1

Concepts

Although Mensor is designed to be intuitive, the nature of the work it performs (metric and measure computation) requires precision and accuracy. As such, it is crucial that users of Mensor know *exactly* how it works, and what assumptions are made in every operation. This resource will cover the core concepts behind mensor. In the *Quickstart* documentation concrete examples are provided.

1.1 Terminology

Mensor uses somewhat standard terminology (derived from statistics and star database schemas), but for clarity we spell out exactly what is meant by these terms in the Mensor universe.

Statistical Unit: The indivisible unit of an analysis, which acts as a single sample in statistical analysis. Examples include: a user, a country, a session, or a document. In this documentation and throughout Mensor, this is used interchangeable with "Statistical Unit Type", "Unit Type" or "Identifier".

Dimension: A feature of a statistical unit which can be used to segment statistical units into groups. Examples include: country of a user, number of hours spent in a session, etc.

Measure: An extensive feature of a statistical unit that can be aggregated across other statistical units of the same type. Note that measures are a subset of dimensions. Examples include: number of hours spent in a session, number of pets owned by users, etc.

Metric: An arbitrary function of measures. Metrics cannot be further aggregated. Examples include: the average number of hours spent in a sessions per user, the average population per country, etc.

Measure Provider: A Python object capable of providing data for a collection of unit types, dimensions, and measures.

Measure Registry: A Python object into which an arbitrary number of measure providers are registered that creates a graph of relationships between providers, unit types and related dimensions and measures, that can then intelligently extract data for any given unit type from all relevant data sources, performing any required joins automatically.

Join: A merging of data associated with a statistical unit from two measure providers.

Partition: A dimension which logically segments data from a measure provider(s) into chunks that can be meaningfully joined. Examples include: the date of the data, which should be used in joins to ensure, for example, that data from a "fact" table in star schema is only ever joined with data from the same date in a corresponding "dimension" table.

Constraint: A condition that must be satisfied for data to included in the result-set of an evaluation.

Evaluation: A computation to generate data associated with a nominated unit type for nominated measures, segmented by nominated dimensions, and subject to nominated constraints.

1.2 The Architecture of Mensor

TBD.

1.3 The Grammar of Measures and Metrics

As it happens, the set of things that one typically wants to do with data in order to generate measures and metrics from data sources is sufficiently restrictive that you can write a grammar for it that is both intuitive and powerful. In this section we explore the key tenets of this grammar.

1) All analyses assume a statistical unit type

While often implicit, it is always the case that for a measure/metric to be meaningful that it must be associated with a particular unit type. Mensor makes this choice of unit type explicit, which allows it to automatically compute relevant statistics (including variance, etc).

2) Joins are always implicit from context

In Mensor, measure providers provide all necessary information to uniquely determine the optimal joins to perform from context. As a result, Mensor never requires the user to perform explicit joins between data from different measure providers. Instead, joins are implicitly performed whenever required. Examples of this are provided in the *Quickstart* section.

3) Unit types can be hierarchical

It is often the case that a unit type can be considered in some sense a subclass of another unit type; for example, users who are also sellers. The grammar adopted by Mensor allows features of more general types to be transitive through to more specific types.

The specifics of this grammar will be explored in more detail in the *Quickstart*.

CHAPTER 2

Installation

If your company/organisation has provided a package that wraps around *mensor* to provide a library of measure and metric providers, then a direct installation of *mensor* is not required. Otherwise, you can install it using the standard Python package manager: *pip*. If you use Python 3, you may need to change *pip* references to *pip3*, depending on your system configuration.

```
pip install mensor
```

Note that this only installs the mensor computation engine, and that you will need to construct your own library of measures and metrics if you use it directly. To get started with this, please review the *Quickstart*.

CHAPTER 3

Quickstart

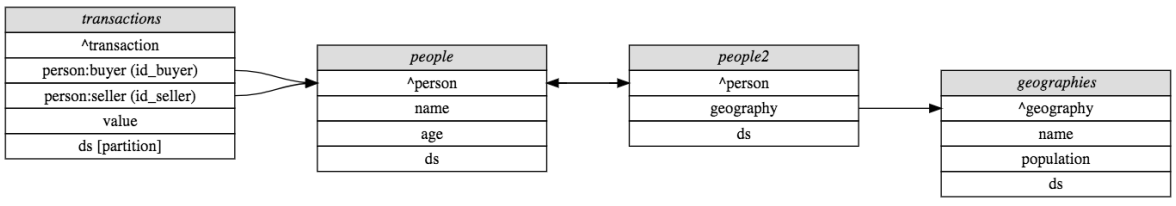
Warning: This resource is in draft status, and missing essential pieces of information, such as outputs of the various examples. This will be remedied once output data types are finalised. The exact API used below is subject to change as the project matures, but the ideas and generic grammar is unlikely to vary. Note also that we do not discuss metrics yet as that API has yet to solidify.

If you have not read the *Concepts* documentation, and find any of the following unclear, be sure to go and read this documentation. If this resource remains unclear, that is no doubt the fault of the author, and you should feel free to reach out by creating a GitHub issue.

In the following, we will explore the use of Mensor in a toy example backed by pandas data frames. All of the code shown on this page can be run equally well in your own Python session if you would like to play with it locally. Note that Mensor is agnostic as to where data is stored, so nothing changes in this API for data stored in database tables.

3.1 Toy Data Schema

The following examples of using mensor will use the following toy schema, which does a reasonable job of demonstrating most of mensor's features.



For the purposes of this tutorial, we will back this schema with pandas dataframes loaded from CSV files, but as far as mensor is concerned, the source of the data is irrelevant. The following Python code sets up a MetaMeasureProvider

which connects to data following the above schema. It is possible to easily set up configuration from YAML files, but we defer such consideration to the *Deployment* documentation.

```
import os
from mensor.measures import MetaMeasureProvider
from mensor.backends.pandas import PandasMeasureProvider

registry = MetaMeasureProvider()

data_dir = "<path to checked out Mensor repository>/tests/data"

people = (
    PandasMeasureProvider(
        name='people',
        data=os.path.join(data_dir, 'people.csv')
    )
    .add_identifier('person', expr='id', role='primary')
    .add_dimension('name')
    .add_measure('age')
    .add_partition('ds')
)
registry.register(people)

people2 = (
    PandasMeasureProvider(
        name='people2',
        data=os.path.join(data_dir, 'people.csv')
    )
    .add_identifier('person', expr='id', role='unique')
    .add_identifier('geography', expr='id_geography', role='foreign')
    .add_partition('ds')
)
registry.register(people2)

geographies = (
    PandasMeasureProvider(
        name='geographies',
        data=os.path.join(data_dir, 'geographies.csv')
    )
    .add_identifier('geography', expr='id_geography', role='primary')
    .add_dimension('name')
    .add_measure('population')
    .add_partition('ds')

)
registry.register(geographies)

transactions = (
    PandasMeasureProvider(
        name='transactions',
        data=os.path.join(data_dir, 'transactions.csv')
    )
    .add_identifier('transaction', expr='id', role='primary')
    .add_identifier('person:buyer', expr='id_buyer', role='foreign')
    .add_identifier('person:seller', expr='id_seller', role='foreign')
    .add_measure('value')
    .add_partition('ds', requires_constraint=True)
)
```

(continues on next page)

(continued from previous page)

```
registry.register(transactions)
```

3.2 The role of MeasureProviders

In the above code, we registered several `MeasureProvider` instances with a `MetaMeasureProvider` instance. Each `MeasureProvider` has the responsibility of being able to provide everything it promised upon request, and we can test this for any particular `MeasureProvider` directly. For example, we can ask the `transactions` measure provider for the sum over all transactions of their value segmented by seller id where the *ds* is '2018-01-01':

```
transactions.evaluate(
    unit_type='transaction',
    measures=['value'],
    segment_by=['person:seller'],
    where={'ds': '2018-01-01'}
)
```

The returned data is a Pandas Dataframe subclass which knows how to keep track of statistics.

Todo: This documentation is incomplete on this point, and will be extended once this component of mensor solidifies.

3.3 Evaluating measures from the MetaMeasureProvider

While it is nice that you can directly evaluate measure from a single `MeasureProvider`, we have not really gained much over just directly accessing the data. Suppose, however, we wish to segment the transaction value measure by sellers' names. Now we need information from multiple providers, and this starts to be a little more taxing if we directly access the data. In Mensor, however, it is as simple as:

```
registry.evaluate(
    unit_type='transaction',
    measures=['value'],
    segment_by=['person:seller/name'],
    where={'ds': '2018-01-01'}
)
```

In the background, mensor is separately asking the *transactions* and *people* data sources for data, and stitching them together for you.

Note: For some backends, such as SQL, dragging down the data locally and doing the joins in memory would be horrendously inefficient. To cater for this use case, :code:'MeasureProvider's have a notion of an "intermediate representation" which they can share with other measure providers that they know to be compatible with themselves. Unless you are deploying Mensor, and need to be aware of such things, this is an implementation detail that is transparent to the user.

Likewise, one might be interested in segmenting the value of transactions and the seller's age by the name of the geography of the seller **and** the buyer's name (admittedly a very contrived example):

```
registry.evaluate(
    unit_type='transaction',
    measures=['value', 'person:seller/age'],
    segment_by=['person:buyer/name', 'person:seller/geography/name'],
    where={'ds': '2018-01-01'}
)
```

Note that mensor also automatically stitched together providers which had the same primary key ("person") in this case.

So far, we have only considered the unit type of 'transaction', but it is also possible to consider other unit types.

What if we want the distribution of transaction values across sellers, segmented by seller name?

```
registry.evaluate(
    unit_type='person:seller',
    measures=['transaction/value'],
    segment_by=['person:seller/name']
)
```

Note the magic that just occurred there. There is no foreign key from *person:seller* to *transaction*, but there is a foreign key from *transaction* to *person:seller*. Mensor took advantage of this to re-aggregate transactions by *person:seller* and the join the resulting sum as a feature of *person:seller*.

Also note that you cannot do the following (because it does not make sense):

```
registry.evaluate(
    unit_type='person:seller',
    measures=['transaction/value'],
    segment_by=['transaction/person:buyer/name']
)
```

This is because it violates the explicit indivisible unit of the analysis (*person:seller*); i.e. a seller may have multiple transactions with different buyers, and so segmenting by any feature of transaction (or its derivatives) would violate the assumption that *person:seller* is the indivisible unit. As such, mensor prevents you from making a statistical faux pas.

3.4 Constraints

There are three principle ways that constraints can be applied, and a rich syntax for specifying the exact contraints.

The constraint application methods are:

- **scoped:** This is the most explicit constraint application method, and allows you to define the constraints that must be enforced even if it is the only reason for accessing a particular measure provider. For example, for unit type transaction: {'person:seller/name': "Matthew"} would restrict transactions to those whose sellers' name was "Matthew", regardless of which measures and segmentations were provided.
- **generic:** This is the most lenient constraint, that only applies if the nominated feature appears in the measure provider being evaluated; but is otherwise silently ignored. Note that it applies *generically* and so will match any measure provider with the nominated field name. For example: {'*/name': 'Matthew'} will filter down to results that have 'name' equal to 'Matthew' for any measure provider that has the field 'name', but is otherwise not enforced.
- **generic for a given unit_type:** This is a cross between the above to methods, which allows a constraint to be enforced whenever a given unit type is being considered, at which point it is enforced and if the unit_type

lacks that feature, an error is thrown. For example: {'*/person:seller/name': 'Matthew'} will enforce that `name == 'Matthew'` every time the current unit type is *person:seller*, but is otherwise ignored.

(Mostly) irrespective of the application method, constraints can be specified in a rich variety of ways. The possible constraint types are:

- **equality:** `{'ds': '2018-01-01'}` implies `[ ds=='2018-01-01']`.
- **inequality:** `{'ds': ('<', '2018-01-01')}` implies `[ ds<'2018-01-01' ]`. The supported operations are: `['<', '>', '<=', '>=']`.
- **in:** `{'ds': {1,2,3}}` implies `[ ds {1, 2, 3} ]`
- **and:** A dictionary or list of dictionaries creates and AND condition, for example: `[{'ds': '2018-01-01', 'name': 'Matthew'}, {'other': 1}]` implies: `[ ds=='2018-01-01' & name=='Matthew' & other==1 ]`
- **or:** A tuple of dictionaries implies an OR condition: `({'ds': '2018-01-01'}, {'other': 1})` implies: `( ds=='2018-01-01' | other==1)`
- **and (nested):** `{'field': [('>', 1), ('<', 2)]}` implies `[ field>1 & field<2 ]`

The types can be nested also, for example: `[({'a': 1, 'b':2}, {'c':3, 'd':4}), ({'e': 5, 'f':6}, {'g':7, 'h':8})]` implies `[ ( [ a==1 & b==2 ] | [ c==3 & d==4 ] ) & ( [ e==5 & f==6 ] | [ g==7 & h==8 ] ) ]`.

Additionally, it is possible to have constraints at different levels in the join hierarchy (for scoped constraints). For example: `({'transaction/value': 100}, {'transaction/person:seller/name': 'Matthew'})` implies `( transaction/value==100 | transaction/person:seller/name=='Matthew' )`.

CHAPTER 4

Deployment

Coming soon.

CHAPTER 5

Contributions

Contributions of any nature are welcome, including software patches, improvements to documentation, bug reports, or feature requests. Be aware, however, that the project is still very much in flux and the original author's full vision has yet to be realised, and so software patches that do not fit into this vision may require significant reworking. If you would like to get involved, it is probably a good idea to open a GitHub issue first, and discuss how best to approach the task.

Welcome! If this is the first time that you have stumbled across this documentation, there is a very good chance you have some questions about this project. That's fantastic! Hopefully, these resources will go some way toward answering them. If you find it lacking in any way, please do not hesitate to file an issue on the GitHub issue tracker.

CHAPTER 6

What is Mensor?

Mensor is a graph-based computation engine for computing measures and metrics. It:

- defines a new grammar for extracting measures and metrics that is designed to be intuitive and capable (it can do almost(?) anything that makes sense to do with metrics and measures).
- makes measure and metric definitions explicit and shareable, and their computations transparent and reproducible.
- allows multiple data sources to be stitched together on the fly without users having to explicitly write the code / query required to join the data sources.
- is agnostic as to how data is stored or accessed, and new data backends are relatively simple to write.
- allows for local ad-hoc definitions of additional data sources for exploration by data scientists or other technically minded folk, decoupling it from deployment into production services.

CHAPTER 7

Why does Mensor exist?

In short, the author (Matthew Wardrop) became frustrated with some (perceived?) operational inefficiencies endemic to the data science industry. In particular, he observed that substantial portions of data science work hours were spent reproducing statistics shown in dashboards, defining ad-hoc segmentations in SQL, and then endlessly debugging them. To make matters worse, despite these efforts taking a significant amount of time, there was little persistence of their efforts beyond their particular analyses, meaning that very similar analyses being done on opposite sides of the company (or done a few months later) all started from scratch. Mensor was created to solve these problems.

CHAPTER 8

How do I use Mensor?

I like where you are going with this line of inquiry! If you are new to Mensor, check out the *Concepts* documentation, and then proceed with the *Installation* instructions. Once installed, you can kickstart your efforts using the *Quick-start* documentation. If you are looking to deploy Mensor as part of a Python package for a team or for production environments, consider exploring the *Deployment* material.

8.1 Indices and tables

- genindex
- modindex
- search