# collate Documentation

***Release 0.3.0***

**DSaPP Researchers**

**Dec 06, 2017**

# Contents

Contents:

# CHAPTER 1

# collate

Aggregated feature generation made easy.

- Free software for noncommercial use: UChicago open source license.
- Documentation: https://collate.readthedocs.io.

**Note**: Collate is now a bundled component of Triage, and future development will take place there. To utilize collate within your custom pipeline, you may still import it from there.

## 1.1 Overview

Collate allows you to easily specify and execute statements like "find the number of restaurants in a given zip code that have had food safety violations within the past year." The real power is that it allows you to vary both the spatial and temporal windows, choosing not just zip code and one year, but a range over multiple partitions and times. Specifying features is also easier and more efficient than writing raw sql. Collate will automatically generate and execute all the required SQL scripts to aggregate the data across many groups in an efficient manner. We mainly use the results as features in machine learning models.

## 1.2 Inputs

Take for example food inspections data from the City of Chicago. The table looks like this:

| inspection_id | license_no | zip | inspection_date | results | violations | ... |
|---|---|---|---|---|---|---|
| 1966765 | 80273 | 60636 | 2016-10-18 | No Entry | | ... |
| 1966314 | 2092894 | 60640 | 2016-10-11 | Pass | ...CORRECTED... | ... |
| 1966286 | 2215628 | 60661 | 2016-10-11 | Pass w/ C... | ...HAZARDOUS... | ... |
| 1966220 | 2424039 | 60620 | 2016-10-07 | Pass | | ... |

There are two spatial levels in the data: the specific restaurant (by its license number) and the zip code. And there is a date.

An example of an aggregate feature is the number of failed inspections. In raw SQL this could be calculated, for each restaurant, as so:

```
SELECT license_no, sum((results = 'Fail')::int) as failed_sum
FROM food_inspections GROUP BY license_no;
```

In collate, this aggregated column would be defined as:

```
Aggregate({"failed": "(results = 'Fail')::int"}, "sum", {'coltype':'aggregate', 'all
↪': {'type': 'mean'}})
```

Note that the SQL query is split into two parts: the first argument to `Aggregate` is the computation to be performed and gives it a name (as a dictionary key), and the second argument is the reduction function to perform. The third argument provides a set of rules for how to handle imputation of null values in the resulting fields.

Splitting the SQL like this makes it easy to generate lots of composable features as the outer product of these two lists. For example, you may also be interested in the proportion of inspections that resulted in a failure in addition to the total number. This is easy to specify with the average value of the *failed* computation:

```
Aggregate({"failed": "(results = 'Fail')::int"}, ["sum","avg"], {'coltype':'aggregate
↪', 'all': {'type': 'mean'}})
```

Aggregations in collate easily aggregate this single feature across different spatiotemporal groups, e.g.:

```
Aggregate({"failed": "(results = 'Fail')::int"}, ["sum","avg"], {'coltype':'aggregate
↪', 'all': {'type': 'mean'}})
st = SpacetimeAggregation([fail],
                                from_obj='food_inspections',
                          groups=['license_no','zip'],
                          intervals={"license_no":["2 year", "3 year"], "zip": ["1␣
↪year"]},
                          dates=["2016-01-01", "2015-01-01"],
                          date_column="inspection_date",
                          state_table='all_restaurants',
                          state_group='license_no',
                          schema='test_collate')
```

The `SpacetimeAggregation` object encapsulates the `FROM` section of the query (in this case it's simply the inspections table), as well as the `GROUP BY` columns. Not only will this create information about the individual restaurants (grouping by `license_no`), it also creates "neighborhood" columns that add information about the region in which the restaurant is operating (by grouping by `zip`). The `state_table` specified here should contain the comprehensive set of `state_group` entities and dates for which output should be generated for them, regardless if they exist in the `from_obj`.

Even more powerful is the sophisticated date range partitioning that the `SpacetimeAggregation` object provides. It will create multiple queries in order to create the summary statistics over the past 1, 2, or 3 years, looking back from either Jan 1, 2015 or Jan 1 2016. Executing this set of queries with:

```
st.execute(engine.connect()) # with a SQLAlchemy engine object
```

will create four new tables in the `test_collate` schema. The table `food_inspections_license_no` will contain four feature columns for each license that describe the total number and proportion of failures over the past two or three years, with a date column that states whether it was looking before 2016 or 2015. Similarly, a `food_inspections_zip` table will have two feature columns for every zip code in the database, looking at the total and average number of failures in that neighborhood over the year prior to the date in the date column. The `food_inspections_aggregation` table joins these results together to make it easier to look at both neighborhood and restaurant-level effects for any given restaurant. Finally, the

`food_inspections_aggregation_imputed` table fills in null values using the imputation rules specified in the `Aggregate` constructor.

## 1.3 Imputation Rules

Imputation rules should be specified in the form of a dictionary:

```
{
    'coltype': 'aggregate',
    'all': {'type': 'mean'},
    'max': {'type': 'constant', 'value': 137}
}
```

The `coltype` key of this dictionary must be one of `aggregate`, `categorical`, or `array_categorical` and informs how the imputation rules are applied.

The other keys of the dictionary are the reduction functions used by the aggregate (such as `sum`, `count`, `avg`, etc.) or `all` as a catch-all. Function-specific rules will take precedence over the catch-all rule. The values associated with these keys are each a dictionary with a required `type` key specifying the rule type and other rule-specific keys.

**Currently available imputation rules:**

- `mean`: The average value of the feature (for `SpacetimeAggregation` the mean is taken within-date).
- `constant`: Fill with a constant value from a required `value` parameter.
- `zero`: Fill with zero.
- `zero_noflag`: Fill with zero without generating an "imputed" flag. This option should be used only for cases where null values are explicitly known to be zero such as absence of an entity from an events table indicating that no such event has occurred.
- `null_category`: Only available for categorical features. Just flag null values with the null category column.
- `binary_mode`: Only available for aggregate column types. Takes the modal value for a binary feature.
- `error`: Raise an exception if any null values are encountered for this feature.

## 1.4 Outputs

The main output of a collate aggregation is a database table with all of the aggregated features joined to a list of entities.

TODO: sample rows from the above aggregation.

## 1.5 Usage Examples

### 1.5.1 Multiple quantities

TODO

### 1.5.2 Multiple functions

TODO

### 1.5.3 Tuple quantity

TODO

### 1.5.4 Date substitution

TODO

### 1.5.5 Categorical counts

TODO

### 1.5.6 Naming of features

TODO

### 1.5.7 More complicated from_obj

TODO

## 1.6 Technical details

CHAPTER 2

# Installation

## 2.1 Stable release

To install collate, run this command in your terminal:

```
$ pip install collate
```

This is the preferred method to install collate, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

## 2.2 From sources

The sources for collate can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/dssg/collate
```

Or download the tarball:

```
$ curl  -OL https://github.com/dssg/collate/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

# Usage

To use collate in a project:

```
import collate
```

CHAPTER 4

# Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

## 4.1 Types of Contributions

### 4.1.1 Report Bugs

Report bugs at https://github.com/dssg/collate/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

### 4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

### 4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

### 4.1.4 Write Documentation

collate could always use more documentation, whether as part of the official collate docs, in docstrings, or even on the web in blog posts, articles, and such.

### 4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/dssg/collate/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

## 4.2 Get Started!

Ready to contribute? Here's how to set up *collate* for local development.

1. Fork the *collate* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/collate.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv collate
$ cd collate/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 collate tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

## 4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, 3.4 and 3.5, and for PyPy. Check https://travis-ci.org/dssg/collate/pull_requests and make sure that the tests pass for all supported Python versions.

## 4.4 Tips

To run a subset of tests:

```
$ py.test tests.test_collate
```

CHAPTER 5

# Indices and tables

- genindex
- modindex
- search