datamode Documentation

Release 0.0.1

datamode

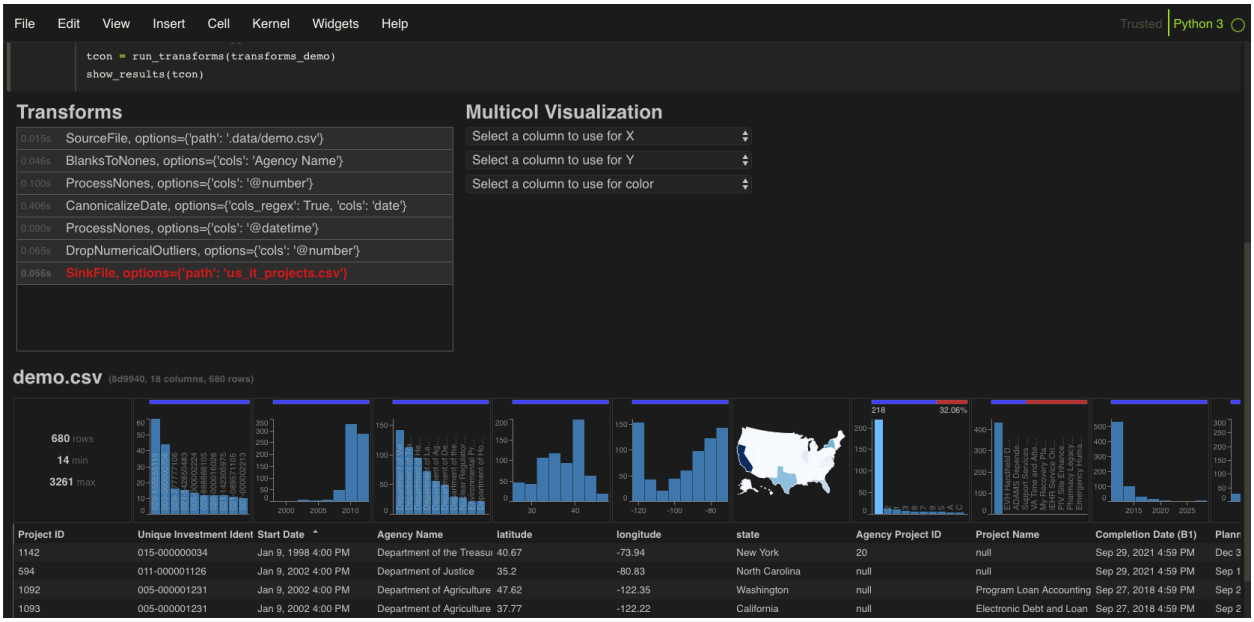
Jan 10, 2019

Contents

Datamode is a free, open source tool to quickly build data science pipelines. Our Python data visualization and transformation tool runs directly in Jupyter Notebook. We help you:

- Explore your data
- Reshape your data
- Deploy your data and models to production.

Here's a screenshot of Datamode running in a Jupyter notebook:



CHAPTER 1

How will this help me?

Getting to know messy data is… well, messy.

Exploratory data analysis is a critical part of any data project but is often a painful and frustrating process. Data engineers and data scientists waste valuable time writing code to load data, extract basic summary statistics, create visualizations, and reshape data. This process is often done in an ad hoc way thus making deployment to a scalable infrastructure or any production like system a time and resource intensive project.

CHAPTER 2

Who is this for?

Anyone and everyone that needs to handle messy data. Our data visualization and inspection tools are built into Jupyter notebook so you can get started in just a few lines of code. We've also built a set of *Transforms* that reduce the amount of code you need to write to get to the same transforms while still being flexible and transparent.

CHAPTER 3

How does Datamode help?

Datamode is built for anyone that needs to quickly undestand data, reshape variables, and push these transforms into a production pipeline. Developers can write and execute code in their Jupyter notebooks and deploy that same code to be used in a scalable environment.

CHAPTER 4

How can I get started?

Check out *Quickstart/Installation* to try Datamode on your own data or you can try a live demo.

If you have any questions, feedback, or issues please e-mail feedback@datamode.com.

Note: Datamode currently requires Python 3.6+ (64-bit). If you're having problems installing, see these pages:

- *Install using pip*
- *Install using conda*
- *Create a Python 3.7 virtualenv*
- *Checking for 64-bit Python*

CHAPTER 5

Feedback / Contributing

If you're trying out Datamode, we'd love to hear from you! We're especially interested in what's working/not working for you, and what's missing that you really need. Email us: feedback@datamode.com.

If you want to contribute transforms or other code, see *Contributing*.

CHAPTER 6

Contents

6.1 Quickstart/Installation

6.1.1 Install using pip

You can install Datamode with pip from your shell/command-prompt:

```
pip install datamode
jupyter notebook
```

Then from your Jupyter Notebook:

```
from datamode.interface import *
quickshow('https://raw.githubusercontent.com/datamode/datasets/master/movies.csv')
```

Play around with the interface a little. Then change the code block to this:

```
from datamode.interface import *

tcon = run_transforms([
  SourceFile('https://raw.githubusercontent.com/datamode/datasets/master/movies.csv',␣
↪sample_ratio=1),
  CanonicalizeDate('release_date'),
  Expression('roi = revenue / budget'),
])
```

See what happens when you click on the histograms or the transforms list. Also, try to delete or change some of the above transforms, or change `sample_ratio`.

You're ready to try our howto guides now: *How To Guide*.

Note: Datamode currently requires Python 3.6+ (64-bit). See below if you have any problems installing.

6.1.2 Install using conda

If you don't have a conda virtualenv, follow these steps:

```
# Create a conda env and enter it - then you can use pip.
# Follow the instructions above to run Datamode in Jupyter.
conda create -n dm_env python=3.7
source activate dm_env
pip install datamode
jupyter notebook
```

If you already have a conda virtualenv, make sure it's using Python 3.6+, because some conda installs run Python 3.5. You can do `import sys; print(sys.version)` to find out what version you're using.

6.1.3 Create a virtualenv

If you aren't already running Python 3.6+, you can create a virtualenv to host Datamode. The easiest way to do that is to install virtualenvwrapper:

```
pip install virtualenv virtualenvwrapper
mkvirtualenv -p python3 dm_env
workon dm_env
```

If you're on windows, install *virtualenvwrapper-win* instead:

```
pip install virtualenv virtualenvwrapper-win
mkvirtualenv dm_env
workon dm_env
```

You'll need to re-enter the virtualenv with `workon` when you start a new shell/command-prompt.

6.1.4 Python architecture (64-bit vs 32-bit)

Datamode only supports 64-bit Python, which you're probably already using. Some versions of conda use 32-bit Python. If you want to make sure you're using 64-bit, you can run this code in Jupyter or any Python environment:

```
import sys
print('64bit' if sys.maxsize > 2**32 else '32bit')
```

6.2 How To Guide

The How To Guide is a set of practical examples of how to use Datamode from setting up the `datamode.yml` to `SourceTable()` and `DropDuplicates()`. For a more problem set driven walkthrough please visit http://www.datamode.com.

6.2.1 Profile Configuration

Datamode uses YAML files to store and organize database connector and config information. Simply get started by creating a file named `datamode.yml` in one of three places:

1. `DATAMODE_CONFIG_DIR` - An environment variable set that contains a directory path to any location you can access.

2. The directory where you plan to execute your transforms.
3. `~.datamode/` - On OSX this is equivalent to your `$HOME` directory and on Windows this can be `$HOME` or `$USERPROFILE` if set. See Python's documentation for more information.

Once you're in the directory you wish to save your `datamode.yml` simply create the file:

```
touch datamode.yml
```

Use your favorite text editor to create the following structure (or copy and paste the below). Be sure to replace `my_connection` with how you want to reference this connection:

```
connections:
  my_connection:
    type:
    host:
    port:
    user:
    password:
    dbname:
    schema:
    default:
```

For example, let's say I would like to create a connection configuration for a MySQL database in `DEV`. I may use the following setup:

```
connections:
  mysql_dev:
    type: mysql
    host: internal-mysql.host.name.com
    port: 3306
    user: dev_user
    password: mypass!!
    dbname: science
    schema:
    default: True
```

You must define `type`, `host`, `user`, and `dbname`. The port will use the database specific default (`3306` for MySQL, `5432` for Postgres).

You can also define `default` to be `True` if you have multiple connection profiles in your `datamode.yml` and do not want to specify it every time you run a `Source` or `Sink`.

The first time you run Datamode a file named `datamode.info` will be saved in either `DATAMODE_CONFIG_DIR` (if set) or `~.datamode/`. This file contains an anonymized unique identifier and expiry that is used for analytics. *You can disable sending anonymized analytics* but you should not delete or alter this file. Altering the file may result in annoying warnings appearing in the logs.

6.2.2 Using Data Sources

Loading in data can sometimes be a hassle. We wanted to make it easier for you to get started with all types of common data sources.

Sourcing Data

Sourcing data is synonymous with loading data. The parent class is `SourceData()` but users should utilize the helper classes listed below. All of the Source features below are possible through `SourceData()` but all parameters

must be explicitly defined.

Sourcing data from SQL

Any interaction with a SQL database requires connection information to be defined. This can be done in your `datamode.yml` (*Setting Profile Configurations*)

Let's assume we have the following `datamode.yml` configured:

```
connections:
  mysql_dev:
    type: mysql
    host: internal-mysql.host.name.com
    port: 3306
    user: dev_user
    password: mypass!!
    dbname: science
    schema:
    default: True
  mysql_prod:
    type: mysql
    host: prod-mysql.host.name.com
    port: 3306
    user: prod_user
    password:
    dbname: science
    schema:
```

If we wanted to source data from `mysql_dev` then we can simply `SourceTable('mydevtable')`. The connection profile `mysql_dev` is used by default since this was set in the yaml.

If we wanted to source data from `mysql_prod` then we would have to explicitly reference it using `conn`. This would look like `SourceTable('myprodtable', conn='mysql_prod')`. Notice how `password` is empty in the yaml. You can supplement an existing connection configuration from a yaml at the time of calling.

Finally, you can also pass a `conn_config` directly to the class without the need for a yaml. `conn_config` accepts a Python dictionary with the keys `type`, `host`, `port`, `user`, `password`, and `dbname` and their corresponding values.

Putting it all together the three examples are as follows:

```
SourceTable('mydevtable'),
SourceTable('myprodtable', conn='mysql_prod', conn_config={'password': 'mypr0dp4s5'}),
SourceTable('anothertable', conn_config={'type': 'mysql', 'host': 'localhost', 'port
↪': 3306, 'user': 'localuser', 'password': 'mylocalpass', 'dbname': 'localdb'})
```

SourceTable

`SourceTable()` loads a complete SQL table. See above for an examples

SourceQueryFile

Datamode supports using a query stored in a separate file. Assume we have a file named *my_query.sql* in the same directory that looks like:

```
SELECT *
FROM mytable
LIMIT 1000;
```

We use this query directly from the file (`dfname` simply names the object so it's referenceable):

```
SourceQueryFile('my_query.sql', dfname='mytable_sample')
```

SourceInline

Users can also inject SQL directly into Source classes using `SourceInline()`:

```
SourceInline('SELECT * FROM mytable LIMIT 1000', dfname='mytable_sample')
```

SourceFile

Datamode supports the loading of CSV and JSONLINES file types. Simply pass the file path to `SourceFile()`. The type of file is inferred from the extension but this can be overridden by passing `file_type` to the class. The `dfname` defaults to the filename without the extension.

Users can also load in JSON or CSV objects directly from a URL.

```
SourceFile('myfile.csv'), # Assumes a CSV
SourceFile('data.logs', file_type='json'), # Assumes a JSON
SourceFile('foo.csv', file_type='json'), # Assumes a JSON
SourceFile('https://raw.githubusercontent.com/datamode/datasets/master/movies.csv'),
↪# Loads a CSV
```

Sinking Data

Sinking data is simply saving it to a location. A rule that is baked into the Sink classes is that the user cannot save back to a table or file that was used as a Source. This is to prevent any accidentally data loss.

SinkTable

`SinkTable()` uses the same mechanics for configuring SQL connections as any of the SQL related Sources. You can Source a table from one database and Sink to another like:

```
SourceTable('mydevtable'), # mysql_dev was set as the default in the YAML,
SinkTable('outputtable', conn='mysql_prod', conn_config={'password': 'mypr0dp4s5'}),
```

SinkFile

`SinkFile()` is similar to `SourceFile()` in that it will try to infer the filetype from the filepath you pass it or this can be overridden by passing `file_type`.

```
SinkFile('ouput.csv'), # Saves as CSV
SinkFile('output.logs', file_type='json'), # Saves as JSON
```

Extracting a DataFrame

The `run_transforms()` method returns a `TransformContext()` object that contains the last table sourced, sinked, or transformed as a DataFrame. It can be retrieved via the `df_current` property. See example:

```
TRANSFORMS = [
  SourceFile('input.csv'),
  SinkFile('output.csv'),
]

tcon = run_transforms(TRANSFORMS)

df = tcon.df_current
```

SetWorkingTable

Datamode applies transforms to the last table sourced, sinked, or transformed. When working with many data tables you may wish to switch back-and-forth for transforms. This can be done by using `SetWorkingTable()`.

```
SourceFile('input.csv'), # dfname defaults to filename without extension
SourceTable('mytable'),
DropDuplicates('unique_ids'), # Drops duplicates in the unique_ids column in the SQL table 'mytable'
SetWorkingTable('input'),
DropDuplicates('dates'), # Drops duplicates in the dates column from the CSV 'input'
```

6.2.3 Using Transforms

Datamode's set of Transforms makes data reshaping much simpler and easier to maintain.

SelectColumns

Let's use the following data table named `ex_data` as an example:

```
# ex_data
uid     first_name     last_name     city            state    country
1       john           smith         san francisco   ca       usa
2       sarah          jenkins       seattle         wa       usa
3       sheila         carey         san diego       ca       usa
```

Running `SelectColumns(['first_name', 'last_name'])` on `ex_data` would return:

```
# ex_data
first_name     last_name
john           smith
sarah          jenkins
sheila         carey
```

DropColumns

Running `DropColumns(['first_name', 'last_name'])` on `ex_data` would return:

```
# ex_data
uid     city            state    country
1       san francisco   ca       usa
2       seattle         wa       usa
3       san diego       ca       usa
```

BlanksToNones

A lot of tools do not recognize empty or whitespace strings as proper NULL values. That's where `BlanksToNones()` comes in to save the day.

Imagine the following data where the first row's `city` field is actually an empty string:

```
# ex_data
uid     city            state    country
1                       ca       usa
2       seattle         wa       usa
3       san diego       ca       usa
```

Applying `BlanksToNones('city')` would convert that empty string into a valid NULL as such:

```
# ex_data
uid     city            state    country
1       NULL            ca       usa
2       seattle         wa       usa
3       san diego       ca       usa
```

ProcessNones

`ProcessNones()` takes in the column(s) you want to process and the action you wish to perform on those `None` values.

Using the following dataset let's run a few different types of `ProcessNones()`.

```
# ex_data
uid     name            price    date
1       tom             NaN      01-03-2018 12:01
2       sally           17.00    01-03-2018 12:01
3       NULL            15.95    01-03-2018 12:01
```

`ProcessNones(['name', 'price'], action='drop')` would drop all rows containing a *NULL*, *None*, *NaN* and yield:

```
# ex_data
uid     name            price    date
2       sally           17.00    01-03-2018 12:01
```

`ProcessNones('price', action='zero')` would replace any *NaN* in the price column and yield:

```
# ex_data
uid     name            price    date
1       tom             0        01-03-2018 12:01
2       sally           17.00    01-03-2018 12:01
3       NULL            15.95    01-03-2018 12:01
```

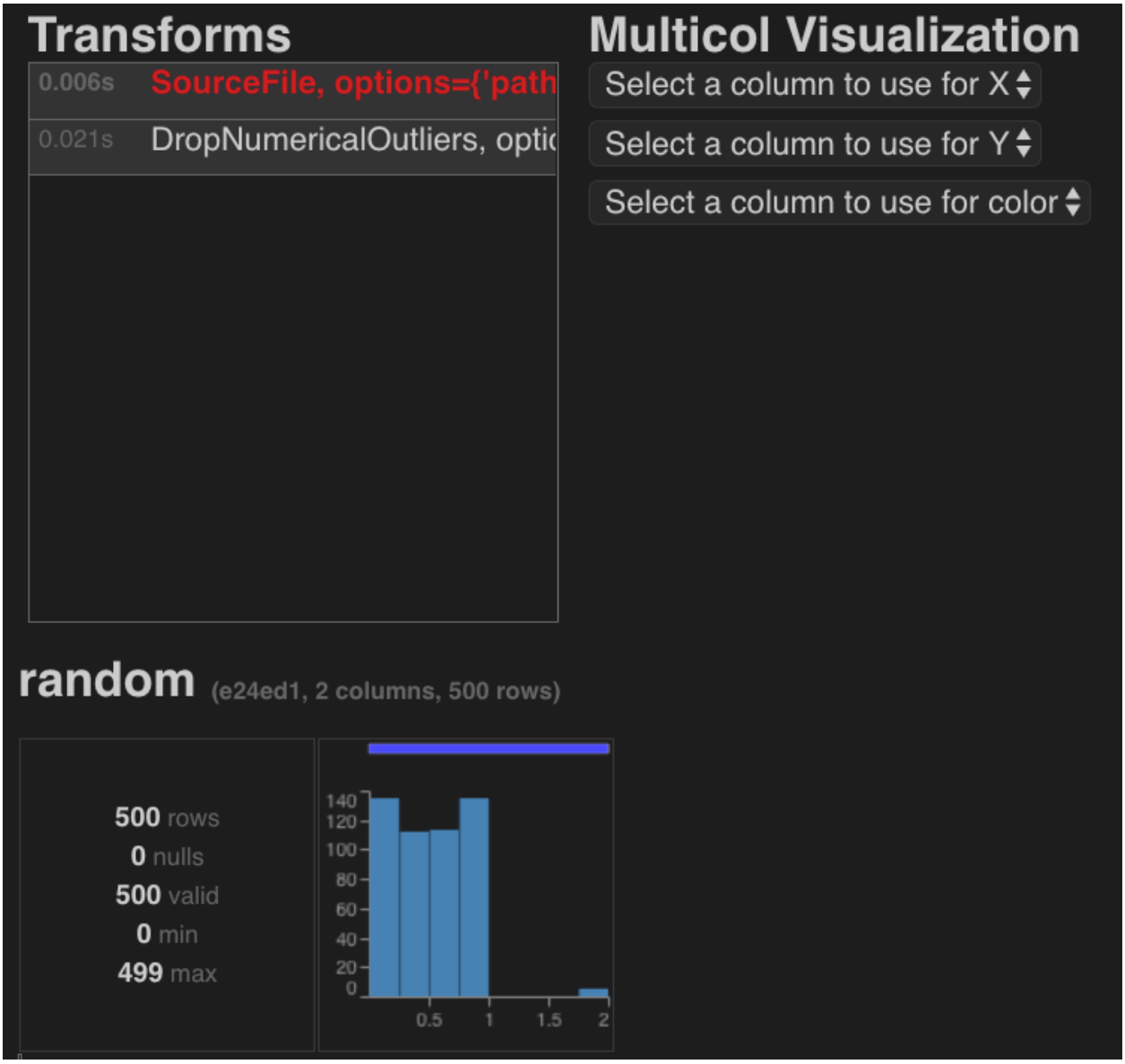
StringToNumber

Sometimes columns containing numerical values are incorrectly cast as strings. Running `StringToNumber()` on the column would convert values to proper integers and any string (such as 'foo') would become `NULL`.

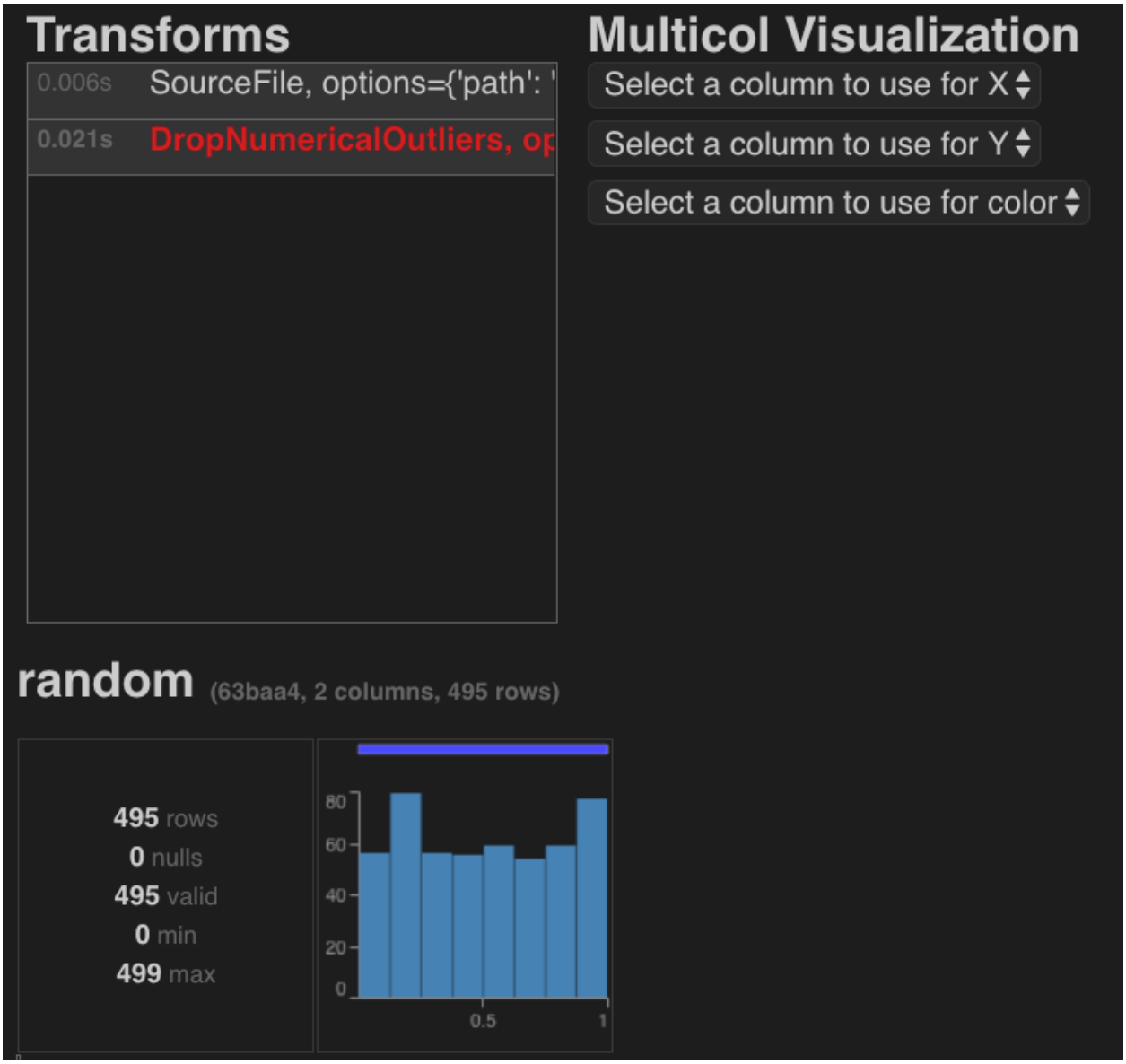
DropNumericalOutliers

`DropNumericalOutliers()` will take a column and drop outliers outside `drop_threshold_mult` times the standard deviations from the average. This is best illustrated in Jupyter notebook:

We have a dataset `random.csv` with a column `foo` containing a mostly uniformly distributed set of numbers except a few outliers as seen in the inline data visualization below:



Running `DropNumericalOutliers('foo', drop_threshold_mult=4)` will drop all values that are 4 standard deviations away from the mean and the resulting data and distribution would look like:



DropDuplicates

`DropDuplicates()` can only be applied to a single column at this time. Please contact us at feedback@datamode.com if you'd like to be able to use more than one column. By default all `None`/`NULL` values will be collapsed into a single record. Set `ignore_nones=True` to preserve all `None` records.

```
# ex_data
uid     name            price    date
1       tom             0        01-03-2018 12:01
2       tom             17.00    01-03-2018 12:01
3       NULL            15.95    01-03-2018 12:01
4       NULL            15.95    01-03-2018 12:01
```

`DropDuplicates('name')` would yield:

```
# ex_data
uid     name            price    date
1       tom             0        01-03-2018 12:01
3       NULL            15.95    01-03-2018 12:01
```

While `DropDuplicates('name', ignore_nones=True)` would yield:

```
# ex_data
uid     name            price    date
1       tom             0        01-03-2018 12:01
3       NULL            15.95    01-03-2018 12:01
4       NULL            15.95    01-03-2018 12:01
```

CanonicalizeDate

`CanonicalizeDate()` will convert strings into Python datetime objects. This is often applied to data with inconsistent date formatting such as:

```
# ex_data
uid     name            price    date
1       tom             0        01-03-2018 12:01
2       tom             17.00    12/31/2017
3       NULL            15.95    01-03-2016
```

`CanonicalizeDate('date')` will normalize all dates into a timezone naive datetime object.

```
# ex_data
uid     name            price    date
1       tom             0        01-03-2018 12:01 PM
2       tom             17.00    12-31-2017 4:00 PM
3       NULL            15.95    01-03-2016 4:00 PM
```

CombineDateAndTimeFragments

`CombineDateAndTimeFragments()` requires the following parameters to be defined

- `col_date` - The original date column.
- `col_time` - The original time column.
- `col_new` - The destination column for the newly created datetime object.
- `consume_originals` - Setting this value to `True` will remove the original date and time columns.

Consider the following example dataset:

```
# ex_data
uid     date          time
1       22/09/2011    14:20:11
2       30/11/2011    10:34:24
3       28/10/2011    14:20:13
```

`CombineDateAndTimeFragments(col_date='date', col_time='time', col_new='new_datetime', consume_orignals=False)` would return:

```
# ex_data
uid     date          time        new_datetime
1       22/09/2011    14:20:11    Sep 22, 2011 7:20 AM
2       30/11/2011    10:34:24    Nov 30, 2011 3:34 AM
3       28/10/2011    14:20:13    Oct 28, 2011 7:20 AM
```

CombineTables

`CombineTables()` is a way to merge two tables. This transform takes the following parameters:

- `merged_table` - the destination table name.
- `tables` - must be a list containing two tables (using `dfname`) that you want merged.
- `relations` - the column you wish to merge the two tables on (this must also be a list).

```
# table1
uid     first_name
1       john
2       sarah
3       sheila

# table 2
uid     last_name
1       smith
2       jenkins
3       carey
```

`CombineTables('full_name_table', ['table1', 'table2'], ['uid'])` would merge table1 and table2 on `uid` and return:

```
# full_name_table
uid     first_name     last_name     _merge
1       john           smith         both
2       sarah          jenkins       both
3       sheila         carey         both
```

NormalizeJson

Columns with `JSON` objects can be flattened using `NormalizeJson()` and new columns will be created from the nested data.

Consider the following data:

```
# json_data
uid     json_data
1       {"city": "San Francisco", "state": "CA"}
2       {"city": "Seattle", "state": "WA"}
3       {"city": "Albany", "state": "NY"}
```

The transform `NormalizeJson('json_data')` would yield:

```
# json_data
uid     json_data                                  json_data.city     json_data.state
1       {"city": "San Francisco", "state": "CA"}   San Francisco      CA
```

(continues on next page)

(continued from previous page)

```
2       {"city": "Seattle", "state": "WA"}          Seattle             WA
3       {"city": "Albany", "state": "NY"}           Albany              NY
```

This will not work with JSON objects in a list or Python dicts.

6.3 Main Interface

6.3.1 Displaying Data

There are two easy ways to get going:

- **`quickshow(input_data)`**
  - Pass either a pandas DataFrame or the name/path of a local csvfile. `quickshow` will display the results in Jupyter Notebook.
- **`run_transforms(transforms, display_results='notebook')`**
  - Runs a stack of `transforms`, which can include sourcing and sinking data. To pass in an existing DataFrame, see `SourceDF`.
  - If `display_results` is set to 'notebook', Datamode will display the UI if it detects that it's running in Jupyter Notebook. Disable this explicitly by passing `None`.
  - Returns a `TransformContext` object. You can access the output DataFrame in the member `df_current`.

6.3.2 Transforming Data

The heart of Datamode is the *Transforms* stack.

- Load your data with SourceFile, SourceDF, SourceSql or any of our *supported* options.
- Choose the transforms you want to apply to your dataset and pass them to `run_transforms`.
- You can then use the DataFrame result (available in `tcon.df_current`), or write to a table or file with `SinkData`.

Run this code in Jupyter Notebook:

```
from datamode.interface import *

tcon = run_transforms([
  SourceFile('https://raw.githubusercontent.com/datamode/datasets/master/movies.csv', sample_ratio=1),
  DropNumericalOutliers('@number'),
  CanonicalizeDate('release_date'),
  SinkFile('output.csv'),
])

# The output DataFrame is tcon.df_current.
```

The above code will fix dates and outliers, write them to output.csv, and visualize your dataset in Jupyter. You can change `sample_ratio` to load more or less of your dataset.

6.4 Data Sources

6.4.1 Configuring Data Connections

Datamode supports loading/saving with the following datastores:

- Postgres, MySQL, local SQLite
- Local csv, json, jsonl files
- S3, GCloud, Azure buckets (planned)
- An existing DataFrame in your Python script

You can add datastore connections to `datamode.yml`, one connection per entry. See *User Profiles* for instructions. When Datamode needs a connection, it will use the default logic below, or you can override the connection with the `conn` parameter to any `Source`/`Sink` transform. You can also override individual options like username, password, etc with `conn_config`. If Datamode needs a connection and doesn't find one, it will error.

Datamode will use connections in this order:

- a connection specified in a transform (e.g. `SourceSql(conn='abc')`)
- a connection marked with `default: true` in the .yml
- a connection if it's the only one in the .yml.

You don't need datamode.yml if you are sending/outputting a DataFrame directly with `SourceDF`, using `SourceFile`, or specifying the connection information manually via `conn_config` to any `Source`/`Sink`.

6.4.2 Data Context

Datamode can work with multiple tables and data connections. By default, Datamode will use the default connection. `Source`/`Sink` transforms can override the connection with the `conn` parameter, and/or override individual values like username/password/etc in `conn_config`.

```
SourceTable('abc')  # Use default connection; current table is 'abc'
SourceTable('def', conn='my_conn')  # Use 'my_conn', set current table to 'def'
DropColumns('some_column')  # Operate on table 'def'
SetWorkingTable(table_name='abc')  # Change table context back to 'abc'.
ProcessNones('another_column')  # Operate on table 'abc'
SinkData('output')  # Write results back to default conn
```

For any transform that operates on a single table, Datamode keeps track of the current working table. This can also be changed with the `table_name` parameter to *SetWorkingTable*.

6.4.3 Datasource Transforms

SourceData

These transforms wrap SourceData and will enforce certain constraints. Any parameter that SourceData accepts can be passed to other Source transforms and will be used if appropriate.

- **`SourceTable(table)`**
  - Loads `table` into Datamode.
- **`SourceQueryFile(queryfile)`**

  - Runs the query in queryfile into Datamode.

- **SourceInline(query)**
  - Runs a sql `query`, for example `select * from abc` and loads the results into Datamode.
- **SourceSql(table=None, query=None, queryfile=None)**
  - Loads the results of a `table`, `query` or `queryfile` into Datamode. If none is specified, Datamode will error. `SourceTable`, `SourceQueryFile`, and `SourceInline` are just convenience wrappers for `SourceSql`.
- **SourceFile(path, file_type=None, sample_ratio=42, sample_seed=42)**
  - Loads a file from `path`.
  - If a URL to a file is passed as a `path` it must begin with http or https.
  - **Options:**
    * `file_type` can be explicitly set or it will be inferred from the extension in the path.
    * `sample_ratio` takes a random sample of your data, to let you work with larger datasets.
    * `sample_seed` lets you fix the random sample deterministically. Pass *None* to get a random sample.
- **SourceDF(df, dfname='dataframe')**
  - Use this when you want to use an existing DataFrame. `dfname` can be overridden.
- **SourceData(dfname=None, table=None, query=None, queryfile=None,path=None, conn=None, c**
  - A generic transform to load data named `dfname` and set the current context. The parameters are the same as the convenience classes above. Any of these parameters (e.g. `conn` or `conn_config` can be overridden in the convenience classes directly. `conn_config` is a dict of options that will override any connection values in datamode.yaml.

SinkData

Sink transforms wrap SinkData. Again, any parameter accepted by SinkData can be used in the Sink transforms if appropriate.

- **SinkTable(table)**
  - Writes `table` to the current sql connection.
- **SinkFile(path, file_type=None)**
  - Writes file `path` with the appropriate output.
  - `file_type` either set explicitly or inferred from path.
- **SinkData(dfname, table=None, path=None, filetype=None, conn=None, conn_config={}, if_e**
  - A generic transform to save data `dfname` to a connection or a file, again using the current context.

Note that to prevent data loss, Datamode will error if you try to write back to any source with the same table or filename that has been sourced. Otherwise, `is_exists` will default to `replace`, and you can also use `append`.

SetWorkingTable

- **`SetWorkingTable(dfname)`**
  - Changes the `dfname`.
  - When using multitable transforms, the tables should be specified explicitly.

Datamode will use the working table on any single-table transforms. The working table is also automatically set with any Source transform. For multi-table transforms, the tables have to be specified explicitly.

6.5 Transforms

6.5.1 Transforming Data

For datasource transforms like `SourceData`/`SinkData`, see *Data Sources*. For the `Expression` transform , see *Expressions*.

Column selection/dropping

- **`SelectColumns(cols)`**
  - Selects columns from the table and drops everything else.
- **`DropColumns(cols)`**
  - Drops columns from the table.

Working with nulls / bad values

- **`BlanksToNones(cols)`**
  - Changes whitespace strings into `None`.
- **`ProcessNones(cols, action)`**
  - Tests for None equivalents (NaT, nan, None, etc), and performs the `action`, either `drop` (drop the whole row) or `zero` (set the value to zero or blank).
- **`StringToNumber(cols)`**
  - Converts all strings in the columns to floats.
- **`DropDuplicates(cols, ignore_nones=False)`**
  - Uniquifies a column by values, by dropping all other rows.
  - Default setting collapses None/NULL/NaN records into one. Set `ignore_nones=True` to preserve all None records.

Distributions

- **`DropNumericalOutliers(cols, drop_threshold_mult)`**
  - Drops numerical outliers outside `drop_threshold_mult` times the standard deviations from the average.

Working with dates

- **`CanonicalizeDate(cols, format=None)`**
  - Converts all string and datetime-string columns into Python datetime columns.
  - You can optionally specify a `format` which will be passed to datetime.datetime.strptime, which can be substantially faster.
- **`CombineDateAndTimeFragments(col_date, col_time, col_new, consume_originals)`**
  - Combines a date column and a time column into a new column `col_new`.
  - Setting `consume_originals=True` will drop the original columns.

Aggregations

- **`CombineTables(merged_table, [tables], [relations])`**
  - Combine `tables` into `merged_table`. The tables will be joined on the first element of `relations`.
  - Only supports one relation right now and one column on each side, however.
  - Both `tables` and `relations` arguments have to each be a list.
- **`NormalizeJson(cols)`**
  - Flattens JSON objects in a column and creates a new column/row for every key/value.
  - New column names will be assigned by using the original column name as a prefix and the JSON key as the suffix.
  - For example, column 'A' has object {"foo": 1, "bar": 2}. This will create two columns named A.foo and A.bar with values 1 and 2, respectively.
- **`Expression(expr)`**
  - See *Expressions*.

In future versions Datamode will have the ability to specify your own transforms or functions that can be run for multiple data levels (cell/row/column/table/multitable).

6.6 Expressions

`Expression` provides a powerful, simple query language for data manipulation, similar to SQL. You can create/replace columns or drop rows, which can use primitives or existing columns in your dataset, including cross-table filters and aggregations.

6.6.1 What can you do with Expressions?

- Arbitrarily complex math operations between columns or for a single column. *Example*
- Create new columns, e.g. set a boolean for any users that are between 25 and 39. *Example*
- Aggregate data from multiple tables - e.g. count all the orders for each user. *Example*
- Easily delete rows that have invalid data, such as user_age < 0. *Example*

- Use a lookup table, e.g. find the income for each user's zipcode from the zipcode table. *Example*
- Add string columns together. *Example*

6.6.2 How to use Expressions

`Expression` is a transform. Pass a single triplequoted string directive to it. Datamode will parse and execute your directive on the dataset.

```
Expression(''' your_expression_here '''),
```

Expression Syntax

```
Full expression:
    <assignment> | <droprows>
    [where ...]
    [join ...]

assignment:
    <tablecolumn> = <formula> [default <primitive>]

droprows:
    drop rows from <[table.]column>

formula:
    <arbitrary nested infix PEDMAS formula including aggregation functions>

where:
    where <tablecolumn> <comparison-operator> (<tablecolumn> | <primitive>)

join:
    join <tablecolumn> <comparison-operator> <tablecolumn>

tablecolumn:
    [tablename.]columnname

aggregations:
    funcname(<formula>)

primitive:
    integer, bool, string, float
```

Note: Some things to keep in mind:

- String literals must be single-quoted.
- Both the table name and column name can separately be optionally double-quoted to allow spaces, e.g. `"my table".mycolumn`.
- All keywords are case insensitive (e.g. aND, WHERE, is null, True, fAlSe, DEFAULT etc).
- You don't strictly need triplequoted strings but it makes writing longer queries easier.
- For now, aggregation functions can only take a single column, nothing else (and only works with joins)
- Don't forget the comma at the end, since `Expression` is just another Transform.

6.6.3 Actions

Assigning columns

You can create or overwrite a single named column by assigning an arbitrary formula to it, which follows standard PEDMAS order. The formula can have any number of nested math expressions, including functions like sum, average and count for cross-table aggregations. Example:

```
Expression(''' mytable.mycolumn = foo / bar + (baz * 3) ** 2 '''),
```

One restriction is that you can't create an infix formula within an aggregation function. This is for performance and complexity reasons. However, if you need this logic, just use two `Expressions` to create an interim column. That will also make your logic and data easier to debug.

Dropping rows

You can use the single `drop rows from <table>` directive. You'll probably want to specify a where clause though, or all rows will be dropped. If you use a join, only rows in the specific drop table will be dropped. Example:

```
Expression(''' drop rows from users
               where user_age < 0 '''),
```

6.6.4 Filtering with `where` clauses

This is similar to SQL - you can compare a column to a primitive, two columns in the same table, or two columns cross-table. Cross-table columns require an explicit join. Don't forget you need to use = for assignment, == (or another operator) for comparison.

Examples:

```
# Compare to a scalar
Expression(''' foo = bar + 1
               where mycolumn > 5 '''),

# Compare to string constant
Expression(''' foo = bar + 1
               where mycolumn == 'somevalue' '''),

# You can use less-than/greater-than since Python does the right thing
Expression(''' foo = bar + 1
               where mycolumn <= 'def' '''),
```

6.6.5 Joining

Datamode currently only supports a single inner join per expression. You can join any two tables on any one column from each. The columns don't need to have the same name. If you need to join more than one table, you should use multiple `Expressions`. That will help you separate and debug your logic as well.

Examples:

```
# Cross-table comparison. Requires a where clause though.
Expression(''' foo = bar + 1
               join users.user_id == orders.order_id '''),
```

(continues on next page)

(continued from previous page)

```
# Joining two different columns which have similar values
Expression(''' foo = bar + 1
               join users.valueA == zipcodes.valueB '''),


# Cross-table where comparison
Expression(''' foo = bar + 1
               where table_1.foo > table_2.bar
               join table_1.my_id == table_2.my_id '''),
```

You can also use this as a lookup engine - see the example below on zipcodes.

6.6.6 Table/column handling

For source columns

You can always specify the table, but you don't need to. If you don't, Datamode will lookup the table from the column. If it doesn't exist, or there are multiple columns with the same name, Datamode will raise an error.

For destination columns

Again, you can always specify the table. Just like source columns, Datamode will attempt to lookup the column. If it doesn't exist, it will be created in the table used by the source columns. If the table doesn't exist, it will be created as well.

6.6.7 Examples

Simple math operations

```
# Simple division
Expression(''' tax = price / 10 '''),

# Exponents and square roots
Expression(''' price_sqrt = price ** 0.5 '''),

# Multicolumn operation
Expression(''' roi = revenue / budget '''),
```

Nested math operations with aggregation functions

You can create an arbitrarily nested expression, but it's often better to break up your `Expression` into multiple `Expressions`, which helps with debugging.

```
# Arbitrarily complex PEDMAS order
Expression(''' newcol = (foo ** bar + baz / (baz * (3 - 2)) ) ** 2 '''),
```

Create multiple columns

Note the comma separating the clauses. The actions share any `where` or `join` clauses.

```
# Two assignments, with a cross-column where clause.
Expression(''' column1 = foo + bar, column2 = bar + baz
               where foo > bar '''),
```

Sum and count total orders per user

Expression can aggregate columns across multiple tables, just like SQL. In this case we're summing and counting the orders for each user, joining the orders and users table. You have to source multiple tables in your transforms, however, e.g. with multiple calls to `SourceFile`, `SourceSql`, etc.

```
Expression(''' users.orders_total = sum(orders.total),
               users.orders_count = count(orders.total)
               join users.user_id == orders.user_id
           '''),
```

Time between signup and first order

We want to find out how long it took the user to make their first order. To do that, we can get the `first` order and subtract the user's signupDate. If any row data is null, the result for that row will be null too.

```
Expression(''' users.time_to_first_order = first(orders.date) - users.signupDate
               join users.user_id == orders.user_id '''),
```

Lookup a value from another table (one to many)

Say you've collected zipcodes for each user. You have another table with income for that zipcode. You can use Expression to do a lookup on the zipcodes table and store the result in the users table.

```
Expression(''' users.zipcode_income = zipcodes.income
               join users.zipcode == zipcodes.zipcode '''),
```

Drop rows

You can remove invalid rows, or really any rows that meet the where expression. You can also join a table to use a cross-table where clause, but rows will only be dropped from the named `drop rows from <table>` table.

```
# Drop obviously invalid rows
Expression(''' drop rows from users
               where users.age <= 0 '''),

# Drop rows with a join
# Will only drop rows from users, not zipcodes
Expression(''' drop rows from users
               where zipcode.income < 10000
               join users.zipcode == zipcodes.zipcode '''),
```

Use spaces or other chars in column or table names

Double quotes are only necessary around table or column names that have spaces or dots.

```
Expression(''' "my new column" = "my table"."my old column" * 2 '''),
```

Create boolean columns from existing columns

With this query, any project valued over 100 will get high priority = true, false otherwise. This can be especially helpful when trying to create dependent variables.

```
Expression(''' projects.high_priority = true default false
               where projects.value > 100 '''),
```

Simple string manipulation

You can use + for simple string concatenation. It won't work with non-strings yet though.

```
Expression(''' address = street_line + ', ' + city + ', ' + state + ' ' + zipcode ''
↪'),
```

Note: For better readability with multiple lines, use Python's triplequoted strings in your expression:

```
Expression(''' mycolumn1 = mycolumn2
               where mycolumn3 > 1 '''),
```

What's not supported yet

- Date to string comparisons (coming soon)
- Sorting, including for aggregations. (coming soon)
- Groupbys, although every join has an implicit groupby on the joined id. (coming soon)
- Operating on strings and non-strings in conjunction
- OR-conjunctions in where clauses
- Python-style string or date formatting
- Arbitrary formulas inside aggregation functions
- Scalar functions like logarithms, etc.
- Intervals like week, date, day, hour, minute etc
- Time windows

If you'd like any of the above to exist, please email us at feedback@datamode.com.

6.7 User Profiles

6.7.1 Configuring Profiles

Users have the option to create a YAML with their datastore connections and other configuration options. See datamode.yml.sample for an example.

Datamode will look for `datamode.yml` in this order:

- the environment variable `DATAMODE_CONFIG_DIR`
- the current Python working directory
- ~/.datamode/datamode.yml

The top-level identifier of datastore connections should be `connections` followed by your connection names and configuration details. For example:

```
connections:
  dev_postgres_db:
    type: psql
    host: localhost
    port: 5432
    user: dev_user
    password:
    dbname: science
    schema:
    default: True
```

Enabling `default: True` to this connection will allow me to Source/Sink data without having to pass the `conn_name`. Users can still pass additional parameters via `conn_config` to Source/Sink functions to supplement parameters in the YAML (e.g. `- conn_config = {'password': 'mypass123'}`). This is particularly useful if you do not want to store sensitive credentials in the YAML itself.

6.7.2 Usage Data

We want to make Datamode an amazing experience for you. To do that, it's important for us to understand how Datamode is being used. So, we collect general usage information every time Datamode is run, like an anonymous unique id. You can see the data that we collect in `src/datamode/utils/usageutils.py` and `$HOME/.datamode/datamode.info`.

We never track credentials, any of the transforms you use, or any of your data.

You can disable this telemetry by modifying your `datamode.yml` profile as shown below. But, please consider leaving it on because it benefits both us and you.

```
connections:
  dev_postgres_db:
    type: psql
    host: localhost
    port: 5432
    user: dev_user
    password:
    dbname: science
    schema:
    default: True
```

(continues on next page)

(continued from previous page)

```
config:
  send_anonymous_usage: False
```

6.8 Contributing

Datamode is an open source project. Feel free to send us a pull request, which we will review under our current license.

We'd love to see what transformations you come up with!

You can always email us at feedback@datamode.com with any suggestions or questions.

6.9 Frequently Asked Questions (FAQ)

6.9.1 Who is this for?

We are data scientists and data engineers, but this library is really for anyone that wants a more elegant solution to exploring data and building pipelines.

6.9.2 How can I use this?

Visit the *How To Guide* to see straightforward examples of using Datamode. For more problem set driven examples check out https://www.datamode.com.

6.9.3 What datasources do you support?

The full list is here: *Data Sources*. An important point is that you can pass a pandas `DataFrame` directly to SourceDF to use it with Datamode, and you can get a `DataFrame` out directly with `tcon.df_current`.