datafuzz Documentation

Release 0.1.2

Katharine Jarmul

Mar 30, 2023

Contents

CHAPTER 1

Motivation

The goal of `datafuzz` is to give you the ability to test your data science code and models with BAD data. Why?? Because sometimes your code will see bad data, especially if you are running it in production. `datafuzz` is motivated by the idea that testing data pipelines, data science code and production-facing models should involve some elements of fuzzing – or introducing bad and random data to determine possible security and crash risks.

CHAPTER 2

Features

- Transform your data by adding noise to a subset of your rows
- Duplicate data to test your duplication handling
- Generate synthetic data for use in your testing suite
- Insert random “dumb” fuzzing strategies to see how your tools cope with bad data
- Seamlessly handle normal input and output types including CSVs, JSON, SQL, numpy and pandas

2.1 Installation

CHAPTER 3

Stable release

To install datafuzz, run this command in your terminal:

```
$ pip install datafuzz
```

This is the preferred method to install datafuzz, 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.

CHAPTER 4

From sources

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

You can either clone the public repository:

```
$ git clone git://github.com/kjam/datafuzz
```

Or download the tarball:

```
$ curl  -OL https://github.com/kjam/datafuzz/tarball/master
```

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

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

4.1 Quickstart

Want to get started right away? Here is a five minute or less tutorial on using `datafuzz`.

CHAPTER 5

Defining YAML Strategies

One of the easiest ways to get started using `datafuzz` is to define strategies in YAML format given a dataset you already have. Let's take a look at an example YAML file:

```
data:
    input: 'file://datafuzz/examples/data/sales_data.csv'
    output: 'file://datafuzz/examples/data/sales_data_with_dupes.csv'
strategies:
    - type: 'duplication'
      percentage: 10
      add_noise: 1
```

In this file, you set up the data `input` and `output`, which are both CSVs. Then, you apply strategies, which is only one item in this case. This calls for duplication of records using 10% of rows and adds random noise to those duplicate rows.

We can now run these transformations from the command line.

```
$ datafuzz run datafuzz/examples/read_csv_and_dupe.yaml
```

When complete, we can check the difference in number of lines for the files.

```
$ wc -l datafuzz/examples/data/sales_data*

2001 datafuzz/examples/data/sales_data.csv
2201 datafuzz/examples/data/sales_data_with_dupes.csv
4202 total
```

That's it! For more information on all available strategies, check out *Strategies*.

CHAPTER 6

Generation and Noise in Jupyter

Are you using Jupyter notebooks for your development? *datafuzz* can easily integrate with your workflow.

To get started, take a look at the example notebooks in the repository.

CHAPTER 7

Generating Synthetic Data

Generating synthetic data to use is easy as Py with `datafuzz`. An easy schema definition can be declared using simple YAML:

```
num_rows: 200
output: 'file:///tmp/iot.csv'
timeseries:
    start_time: 2015-01-01T00:00:00
schema:
    username: faker.user_name
    temperature: range(5,30)
    heartrate: range(60,90)
    build: faker.uuid4
    latest: [0, 0, 1]
    note: ['interval', 'sleep', 'wake', 'update', 'user', 'test', 'n/a']
```

This file declares some useful schema, such as the number of rows to generate (`num_rows`), timeseries information (which is only required if you want a timeseries to be generated) and the schema for each row. You can use ranges, aranges, lists or faker providers (see faker provider documentations).

To generate the data, you can run the command line:

```
$ datafuzz generate datafuzz/examples/yaml_files/iot_schema.yaml
```

To see our generated data, we can peek at the output file:

```
$ head -n 5 /tmp/iot.csv

build,temperature,username,heartrate,latest,note
59803106-7fa4-5fe3-2ad8-0e962c4e5666,13,rramirez,86,0,n/a
d865fbc7-d43a-e001-ea67-d1892c26aa41,26,kristi42,72,0,n/a
535f4f08-ca2b-c418-081b-bc8e572087e9,7,jacksonterri,88,1,n/a
69e2796f-f2a2-f139-1b06-cbc500cb387b,6,eerickson,75,0,wake
```

7.1 Frequently Asked Questions

1. Why would I want to mess up my dataset?

   `datafuzz` is not to be used for every data science problem, but there are several where adding noise, nulls, fuzz or duplication can help you test and determine the resiliency of your model, pipeline or data processing script. It is built with these use cases in mind, so you can break your code before someone does it (intenionally or otherwise).

2. Why not use `Hypothesis`?

   Why use just one? `Hypothesis` is a great tool, which I recommend for all data scientists for property-based testing. But hypothesis is not a tool for adding noise to an already compiled (or synthetically compiled dataset). Hypothesis can be used to generate a series of property-defined examples for your pipeline or use case; however, if you want to test for unexpected types or for realistic looking noise using your already defined dataset, it becomes difficult and cumbersome. This is one of the reasons I originally built `datafuzz`. For this reason, I think it is useful to have more than one tool for your data science testing needs.

3. Why doesn't `datafuzz` have X feature?

   It's likely I didn't think it should be included in the initial scope. That said, I am all for determining good future features and welcome well-described and simple requests (as well as pull requests!). Head on over to the GitHub Issues to see if the feature is already in the works or open a new Issue to start the conversation. For more details on contributing, see the contributing guide.

4. What is fuzzing? Why use it in data science?

   Fuzz testing tests bad or malicious inputs and determines if the program crashes or raises errors. It is often used in the security community to investigate potential risks like buffer or stack overflows. Why use fuzzing for data science? Like software, data science code is often exposed to user input or outside APIs. Because of this, it is vulnerable to some of the same issues and attacks seen in web services. Even if the attacks or bad data are not malicious or are created by a bug in an internal system, we should test how the data science application, code or model behaves when given corrupt, noisy or duplicate data. Even if the expected behavior is a crash, we should know and test that in advance. This also helps determine if your extraction (ETL) or processing code (pipelines and workflows) address the issues or raise warnings when they see unexpected values.

Read More:

7.1.1 Usage

To use datafuzz in a project, you have multiple options. You can use the Command Line Interface or with your normal Python script or Jupyter notebooks.

7.1.2 CLI

The easiest and fastest way to get started using `datafuzz` is via the command line interface, or CLI. There are a few ways to do so. First, you should determine if you need to generate data or simply modify data you have.

Generate command

If you need to generate data, you should use the cli `generate` command. This has two options:

- Utilize a YAML file which defines the schema for your synthetic data.

- Pass descriptions in via command line flags (not recommended for long or complex schema as this is not easily maintainable).

A good example of the YAML usage is included in the quickstart.

Let's take a look at how to use the command line flags:

```
$ datafuzz generate --non-yaml -h
    usage: datafuzz [-h] [-f FIELDS] [-v VALUES] [-o OUTPUT] [-n NUM_ROWS]
                    [--start_time START_TIME] [--end_time END_TIME]
                    [--increments {hours,seconds,days,random}]
                    {generate}

      Generate dataset: to use

      positional arguments:
            {generate}

      optional arguments:
            -h, --help            show this help message and exit
            -f FIELDS, --fields FIELDS
                            semicolon-delimited string of field names
            -v VALUES, --values VALUES
                            semicolon-delimited string of values.This can be a mix
                            of faker types and ranges
            -o OUTPUT, --output OUTPUT
                    what output to use
            -n NUM_ROWS, --num_rows NUM_ROWS
                    number of rows to generate
            --start_time START_TIME
                    start time of timeseries in isoformat:YYYY-MM-
                    DDThh:mm:ss
            --end_time END_TIME   end time of timeseries in isoformat: YYYY-MM-
                    DDThh:mm:ss
            --increments {hours,seconds,days,random}
                    how to increment entries
```

To specify we aren't using YAML we pass a `--non-yaml` flag, which allows us to access the CLI parsers. For generation, we see a long list of possible options, let's try a few!:

```
$ datafuzz generate -f 'name;age;city' -v 'faker.name;range(30,40);faker.city' -n 200␣
↪-o file://friends.csv

dataset now available at friends.csv
```

Now let's check the content:

```
$ head -n 5 friends.csv

name,city,age
Eric Walsh,West Brandy,36
Jason Willis,Port Stephen,37
Kyle Greer,North Brandon,32
Mathew Ward,North Ginabury,32
```

That was easy! :)

For a review of all options you can use with the `generate` command, check out the *Generators*.

Run command

A second option might be that you want to modify data you have or data you just generated. To do so, you can use the `run` command. Similar to the `generate` command, this has two option:

- Utilize a YAML file which defines the different transformations to run on your data
- Pass a type of run directly into the command line (and repeat as needed)

A good example of the YAML usage is included in the quickstart.

Let's take a look at run with just command line options:

```
    $ datafuzz run --non-yaml -h

usage: datafuzz [-h] [-i INPUT] [-o OUTPUT] [-s STRATEGIES] [--db_uri DB_URI]
                  [--query QUERY] [--table TABLE]
                  {run}

  Apply datafuzz strategies to input, return output

  positional arguments:
    {run}

  optional arguments:
    -h, --help            show this help message and exit
    -i INPUT, --input INPUT
                          input string (filename or sql)
    -o OUTPUT, --output OUTPUT
                          input string (filename or sql)
    -s STRATEGIES, --strategies STRATEGIES
                          dictionary defining the strategies to take
    --db_uri DB_URI       If using database, the db URI to connect
    --query QUERY         If using db input, query to collect data
    --table TABLE         If using db output, table to insert into
```

Okay, let's give it a shot with our newly generated `friends.csv` file:

```
    $ datafuzz run -i file://friends.csv -o file://fuzzy_friends.csv -s '{"type":
↪"fuzz", "percentage": 30}'

dataset now available at fuzzy_friends.csv
```

And we can check our output:

```
$ head -n 5 fuzzy_friends.csv

    ,name,city,age
    0,Eric Walsh,b'\xef\xbb\xbf'West Brandy,36
    1,Jason Willis,Port Stephen,37
    2,Kyle Greer,North Brandon,32
    3,Mathew Ward,North Ginabury,32
```

And indeed, our friends now have some fuzz! For a review of all options you can use with the `run` command, check out the *Strategies*.

For a more in-depth look into `datafuzz`, see *Developer Interface*.

7.1.3 Using datafuzz with Python or Jupyter

You don't need to use `datafuzz` with the CLI, you can also use it with your native Python scripts, frameworks or Jupyter notebooks. To see some Jupyter notebook integration examples, check out the Jupyter Notebooks included in the examples directory.

For integration with your Python script, the necessary parameters for initialization may differ depending on the class you are using to transform your data.

To do so, you might start with a dataset in the shape of a Pandas DataFrame or a *numpy* matrix or even a Python list of dictionaries and list. You could also generate a new dataset by using the generator class.

Let's generate a simple timeseries using the generator:

```
from datafuzz.generators import DatasetGenerator

generator = DatasetGenerator({
    'output': 'pandas',
    'schema': {
        'category': list('ABCD'),
        'model': range(4,8),
        'plate': 'faker.license_plate',
        'year': range(2001, 2018),
        'color': 'faker.safe_color_name',
        'price': range(20000, 50000, 1000)
    },
    'num_rows': 1000,
})

generator.generate()

dataset = generator.to_output()

print(dataset.head())
```

Your output should look something like this:

```
  category    color  model     plate  price  year
0        A  fuchsia      4   0736 CF  20000  2003
1        B     teal      6   EXS 036  29000  2004
2        D     teal      6   1QX5388  32000  2009
3        C     navy      5  6P 15774  30000  2011
4        A    white      4   0SQ D88  31000  2013
```

Now we have a dataset that holds our generated dataframe. If instead we had imported or transformed the data into a dataframe, we can start at this step.

Now that we have some data to work with, let's determine what transformation there are available. The strategies available are the following classes:

- Duplicator
- Fuzzer
- NoiseMaker

Let's use the `NoiseMaker` class to add some noise to our dataset.

```
from datafuzz import DataSet, NoiseMaker
```

(continues on next page)

(continued from previous page)

```
dataset = DataSet(dataset,
                  output='file://my_new_file.json')

noiser = NoiseMaker(
    dataset,
    noise=['add_nulls', 'random'],
    columns=['price', 'model', 'year'],
    percentage=30,
)
noiser.run_strategy()
```

At this point, your `DataSet` object is transformed. You can check it by looking at the 5 initial items in the dataset:

```
print(dataset[:5])
```

Your data should now be a *bit* messy:

```
category    color     model     plate         price         year
0        D   maroon  4.376930   179-IYJ  29000.000000  2006.000000
1        D    green  4.000000  P 336983  15468.136372  1598.702067
2        D     gray  5.270262   DIV-042  20000.000000  2002.000000
3        C     aqua  2.017815   84R 707  38000.000000           NaN
4        C  fuchsia  6.000000   8078 TU  37000.000000  1995.355014
```

You can continue running transformations, if you like:

```
from datafuzz import Duplicator

duplicator = Duplicator(
    dataset,
    percentage=20,
)
duplicator.run_strategy()
```

When you are done with the transformations, you can export the data depending on the output you set when the dataset was first initiated. You can also set a new output string. Available outputs are as follows:

- pandas dataframe: 'pandas'
- numpy 2D array: 'numpy'
- `datafuzz.DataSet`: 'dataset'
- list: 'list'
- CSV files: 'file://foo.csv'
- JSON files: 'file://foo.json'
- SQL: 'sql'

If you use the 'sql' output, you need to also set a value for `'table'` and for `'db_uri'`. For an in-depth treatment of input and output options, please see *I/O Options*.

To then get the output, you need to run `to_output`:

```
output = dataset.to_output()

print(output)
```

And now to check the file:

```
head -c 200 my_new_file.csv

{"0":{"category":"D","color":"maroon","model":2.5774000851,"plate":"0ME 062","price
↪":null,"year":2008.0},"1":{"category":"C","color":"black","model":null,"plate":"UGV-
↪266","price":39000.0,"year":2010.
```

That covers the vast majority of the functionality contained within *datafuzz*. Want to see more features? Check the backlog and feel free to follow steps for contributing!

If you want to see more examples of functionality, check out the examples in the repository.

For a longer explanation of the generators and their functionality, see the *Generators* documentation. For the same treatment for strategies, see the *Strategies* documentation.

For a review of I/O functionality and options (for both parsers and `DataSet` output), see the *I/O Options* documentation.

7.1.4 Generators

Currently, there is only one generator `DatasetGenerator` which allows you to define schema for new datasets. This generator requires use of the `faker` library <http://faker.readthedocs.io/>‘_.

Generator Initialization Requirements

To initialize a generator you need the following:

parser: a `parsers.SchemaYAMLParser` or `parsers.SchemaCLIParser` or a dictionary with the appropriate keys (see Required Parser Arguments below)

output: a string specifying output (see: *I/O Options*)

Required Parser Arguments

The `SchemaYAMLParser` and `SchemaCLIParser` as well as any dictionary you use in leiu of a parser object need to have certain keys in order to generate the data. The required arguments are as follows:

schema: a dictionary of column names and values to use. Optional values are any of the faker providers defined like `'faker.name'`, as well as lists or iterators of options or `range` and `arange` objects.

num_rows: an integer for the number of rows to generate

The parser objects (or a dictionary you use) may also have the following optional arguments which are used to create timeseries:

start_time: isoformat start time which will be used to generate a timeseries. Currently, the timeseries column defaults to ‘timestamp’.

end_time: if desired, set an end_time in isoformat. Note: this may or may not be reached given the number of rows requirement.

increments: choice from: ‘days’, ‘hours’, ‘seconds’ and ‘random’ for timestamp increments. Default is random which is a mix of days, hours and seconds.

For more examples on how to utilize these generators, check the *Usage* documentation.

7.1.5 Strategies

In `datafuzz`, strategies are used to define ways to fuzz or add noise to data. There are currently three types of strategies which reflect three different classes: `Duplicator`, `NoiseMaker` and `Fuzzer`.

Required Initialization Values

To use each strategy, you must define certain attributes, as follows.

For **all strategies**, you need to define:

dataset: a `datafuzz.DataSet` object to apply the strategy to

percentage: percentage of rows to fuzz, noise or duplicate (0-100)

The `NoiseMaker` class has some additional requirements:

columns: a list of columns to apply the noise to (this will be chosen at random if not provided)

noise: a list of possible noise to apply. Options are:

- 'add_nulls': add null values
- 'string_permutation': apply string transformations
- 'random': generate some random values based on col type
- 'range': change values into given or column range
- 'type_transform': apply type transformations

The `Fuzzer` class has one additional requirements:

columns: a list of columns to apply the fuzz to (this will be chosen at random if not provided)

The `Duplicator` class has one additional options:

add_noise: boolean to signify if random noise should be applied to the duplicated rows

Running the strategy

For each strategy class, you can run the strategy using `self.run_strategy`. This will apply the transformation directly to the dataset records.

7.1.6 I/O Options

Whether you are using a generator or a strategy, you will want to define inputs and outputs for your data. These are standard across `datafuzz` as they are defined in the `DataSet` class. The current supported data types for a dataset are:

- lists
- numpy 2D arrays
- pandas dataframes

`datafuzz` will utilize pandas dataframes internally to represent and modify the records if you have pandas installed. If you want to avoid this, you may pass `pandas=False` in your initialization of your `DataSet` object.

Input options

You can read in several additional data formats, which will be used to create a `DataSet` object. These are normally defined in the Parser object, or are passed in the `DataSet` object itself. Options are as follows:

files: defined by specifying `file://$PATH_AND_FILENAME`. Currently, only CSV and JSON files are supported.

sql queries: defined by passing `'sql'` as input. You must then also pass optional arguments for your parser (`db_uri` and `query`)

Output options

Output can be generated from every `DataSet` object by calling the `to_output` method, which will return either the output string or the output object. It will return a string for non-Python objects (such as sql tables and files) and an object for all native objects.

For output, you can define the following options:

files: defined by specifying `file://$PATH_AND_FILENAME`. Currently, only CSV and JSON files are supported.

sql table: defined by passing `'sql'` as output. You must then also pass optional arguments for your output (`db_uri` and `table`)

pandas dataframe: defined by passing `'pandas'`

numpy 2D array: defined by passing `'numpy'`

list: defined by passing `'list'`

If you are interested in an example of using `datafuzz` as a stream, please see the streaming example in the example directory.

7.1.7 Developer Interface

Here are the main interfaces of datafuzz for general use.

Dataset class

class datafuzz.**DataSet**(*input_obj*, ***kwargs*)

DataSet objects are used as the primary datatype for passing around data in datafuzz.

If pandas is installed, it will use dataframes to load and transform data; otherwise, it will use a list. You can also specify to not use pandas by passing keyword argument *pandas=False*.

Supported inputs are JSON and CSV files, numpy 2D arrays, sql queries (you must pass a *db_uri* keyword argument and a *query* argument), pandas DataFrames and Python lists (of dictionaries or lists).

Attributes: DATA_TYPES (str): list of possible datatypes (pandas, numpy, list). FILE_REGEX (str): regex to find file name USE_PANDAS(bool): boolean that determines whether pandas is

installed and also OK to use (no *pandas=False*)

records (list): data records for input (obj): initial input for dataset

(can be dataframe, list, numpy array, filename or *sql*)

output (str): output

(if specified, can be dataframe, list, numpy array, filename or *sql*)

original (obj): copy of input which won't be modified data_type (str): dataset datatype (pandas, numpy, list). db_uri (str): dataset database connection string

(required only if using *sql* as input or output)

query (str): dataset database select query string (required only if using *sql* as input)

table (str): dataset database output table (required only if using *sql* as output)

append(*rows*)

Append rows to DataSet records

Arguments: rows (list): rows to add or concatenate

TODO:

- is a shuffle needed?
- should the index be maintained or reordered
- should new indexes be ordered or not

column_agg(*column*, *agg_func*)

Perform aggregate function on given column

Arguments: column (int): column index agg_func (function): aggregate function to perform on column

Returns aggregate result

Example:

dataset.column_agg(3, min)

column_dtype(*column*)

Return dtype of column

Arguments: column (int): column index

Return: data type of the column

TODO:

- determine smart way to test more than one row for a list?

column_idx(*column*)

Return numeric index of a column

NOTE: if column is not found, raises an AttributeError

input_filename

Return filename if input follows proper file format file://{[}absolute or relative filepath]

NOTE: this will raise an exception if the file is not found

output_filename

Return filename if output follows proper file format file://{[}absolute or relative filepath]

sample(*percentage*, *columns=False*)

Get a sample from the dataset.

Arguments: percentage (float): percentage of dataset to sample should be a value from 0.0-1.0

Kwargs: columns (bool): option to sample columns from dataset default is False

Returns: A sample from the dataset with matching datatype

to_output()
Transform DataSet records to output. This uses helper method *obj_to_output* located in *output/helpers.py*

Returns output object or filepath.

validate_db()
Validate that proper variables are set and a connection can be established with the database if either input or output are set to *sql*.

This will raise an exception if validation fails.

validate_parsed()
Validate if data was properly parsed. This tests:

- valid data types
- records properly parsed and set to self.records
- self.original exists

It will raise an exception if the validation fails.

Strategy classes

class datafuzz.strategy.**Strategy**(*dataset*, ***kwargs*)
Strategy objects apply predefined noise and fuzz to datasets.

Parameters: dataset (*datafuzz.DataSet*): dataset to noise / alter

Kwargs:

percentage (int) [percentage to distort (0-100)] If none given, default to 30

Attributes: dataset (*datafuzz.DataSet*): dataset to noise / alter percentage (float) : percentage to distort (0-1)

NOTE: each strategy type may have additional required keyword arguments

see also: *duplicator.Duplicator*, *noise.NoiseMaker* and *fuzz.Fuzzer*

apply_func_to_column(*function*, *column*, *dataset=None*)
Apply a function to a column in a given dataset.

(this should work as uniformly as possible across data types)

Arguments: function (lambda or other func): function to apply column (int): column index

Kwargs:

dataset (*dataset.DataSet*): dataset to use defaults to self.dataset

Returns: None

Note: This performs transformations on *dataset.records* in place.

get_numeric_columns(*columns*)
Ensure columns are numeric, this will get indexes of string column names (i.e. Pandas columns or dict keys)

Arguments: columns (list of str or int): column list

Returns: columns (list of int): column list (only ints)

num_rows
return number of rows to transform in dataset based on given percentage.

NOTE: this uses rounding so only whole numbers are returned.

class datafuzz.duplicator.**Duplicator**(*dataset*, ***kwargs*)

Duplicator is used to duplicate rows in a dataset

see also: *strategy.Strategy*

noise(*sample*)

Adds noise to the duplicate rows

Parameteres: sample (list or obj): *dataset.Dataset.sample*

Returns sample (list or obj): distorted rows

TODO:

- implement more noise options than just random

run_strategy()

Run duplicator strategy and if add noise is selected, add noise to the data before appending it to the dataset.

class datafuzz.fuzz.**Fuzzer**(*dataset*, ***kwargs*)

Fuzzer is used as a strategy to add "dumb" fuzzing methods (i.e. random bad values). These transformations are mainly based on column type.

see also: *strategy.Strategy*

fuzz_date()

Return random choice from date fuzz helpers.

Possible transformations:

- shift_time: shift the time by a random amount
- date_to_str: transform to string

fuzz_numeric()

Return a random choice from the numeric fuzz helpers.

Possible transformations:

- nanify: insert null values (sometimes strs)
- bigints: return big magic numbers
- hexify: return hex value

fuzz_random()

Return a random choice from the random fuzz helpers.

Possible transformations:

- sql: returns unkind sql
- metachars: inserts metacharacters
- files: returns filepaths or bash
- delimiter: inserts multiple delimiters
- emoji: inserts one random emoji

fuzz_str()

Return random choice from string fuzz helpers.

Possible transformations:

- add_format: insert format strings
- change_encoding: decode with possibly bad encoding

- to_bytes: transform to bytes
- insert_boms: insert utf-8 boms

run_strategy()
Apply fuzz methods to chosen columns.

For now, this applies a mixture of random and column type based transformations.

See *Fuzzer.fuzz_str*, *Fuzzer.fuzz_random* and *Fuzzer.fuzz_numeric* for full list of possible transformations.

class datafuzz.noise.**NoiseMaker**(*dataset*, ***kwargs*)
NoiseMaker applies noisy data transformations to given dataset.

see also *strategy.Strategy*

nullify()
Set null values for sample in columns

randomize()
Set random values for sample in columns

NOTE: this will vary based on column type

run_strategy()
Run noise strategy on sample

Performs transformations on self.dataset

set_value(*value*, *column=None*)
Set value for a series of columns or one column.

Arguments: value (obj): value to set

Kwargs: column (str or int): name or index of column

TODO:

- should this be available on Strategy class?

string_permutation(*column=None*)
Permute string values for sample in columns

TODO:

- add permutations for missing characters
- flipped strings
- typos
- homonyms / autocorrect

type_transform()
Transform types for sample in columns.

NOTE: if a string column is used and the values cannot be transformed into integer or float values, you may not see a useful transformation.

TODO:

- for strings, should numeric values be inserted as strings

instead?

use_range()
Use values from a range to set values in columns

If *limits* not passed during initialization, this method will attempt to determine good limits based on the column ranges and use those.

NOTE: range is only available for numeric columns

TODO:

- should we calculate IQR and insert outliers?
- if not, should add_outliers be a new option for noise?

Parser classes

class datafuzz.parsers.**StrategyYAMLParser**(*file_path*)

Strategy YAML Parser is used to parse strategies and fuzz / transform data using a simple YAML definition.

see also *parsers.core.BaseYAMLParser*

db_uri
Return data db_uri from parsed YAML

execute()
Execute strategies from parsed YAML

input
Return data input from parsed YAML

output
Return data output from parsed YAML

query
Return data query from parsed YAML

strategies
Return strategies from parsed YAML

table
Return data table from parsed YAML

class datafuzz.parsers.**StrategyCLIParser**(***kwargs*)

Strategy YAML CLI is used to parse strategies and fuzz / transform data using a simple CLI definition.

execute()
execute fuzzing strategies from parser

Returns: output

init_parser()
Initialize parser with required and optional arguments

Returns: argparse.ArgumentParser

parse_args(*argv=None*)
Parse arguments and validate them

Kwargs: argv (sys.argv or similar list)

print_help()
print parser help

validate_arguments()
Validate that all required fields are submitted

class datafuzz.parsers.**SchemaYAMLParser**(*file_name*)
Schema YAML Parser is used generate data using a simple YAML definition.

see also *parsers.core.BaseYAMLParser*

execute()
generate data using parsed YAML

Returns: output

num_rows
Return num_rows from parsed YAML

output
Return output from parsed YAML

parse_timeseries()
Parse and set values related to timeseries

raises SyntaxError if start or end time were not properly parsed

schema
Return schema from parsed YAML

timeseries
Return timeseries from parsed YAML

validate_yaml()
Validate that all required fields are parsed from YAML

raises SyntaxError if required field missing

class datafuzz.parsers.**SchemaCLIParser**(***kwargs*)
Schema Parser for CLI Input This generates a argparser to parse input and can be used to then generate the dataset

execute()
Generates data from CLI parsed arguments

Returns: output

init_parser()
Generate *argparse.ArgumentParser* to use for parsing arguments

parse_args(*argv=None*)
Parse arguments and validate them

Kwargs: argv (sys.argv or similar list)

print_help()
print parser help

validate_arguments()
Validate that all required fields are submitted

Output classes

class datafuzz.output.**CSVOutput**(*dataset*, ***kwargs*)
CSV output for writing datasets to CSV file.

see also: *datafuzz.output.BaseOutput*

to_csv()
Write the CSVOutput to a csv file

class datafuzz.output.**JSONOutput**(*dataset*, ***kwargs*)
JSON output for writing datasets to JSON file.

see also: *datafuzz.output.BaseOutput*

to_json()
Write the JSONOutput to a json file

class datafuzz.output.**SQLOutput**(*dataset*, ***kwargs*)
Database output for writing datasets to a table.

see also: *datafuzz.output.BaseOutput*

Extra parameters: db_uri (str): Database URI String table (str): Database table name

to_sql()
Write the dataset records to a sql table

datafuzz.output.**obj_to_output**(*obj*)
Transform DataSet or generator records to output

supported outputs: dataset, pandas, numpy, list, csv and json (specify file://\protect\T1\textdollarNAME.csv or file://\protect\T1\textdollarNAME.json) and sql (specify db_uri and table)

NOTE: will raise exception if unsupported output set

Generator classes

class datafuzz.generators.**DatasetGenerator**(*schema_parser*)
DatasetGenerator creates a dataset when given a parsed YAML or series of CLI arguments or passed arguments.

Attributes:

parser (*parsers.SchemaYAMLParser*, *parsers.SchemaCLIParser* or dict): parsed YAML, CLI or

dict with necessary keys

output (str): output description num_rows (int): number of rows to generate timeseries (bool): whether to generate a timeseries records (list): generated data fake (faker.Faker): Faker object to generate data

Parser parameters:

schema (dict): dictionary of column names and values to use (i.e. {'foo': 'faker.name',... })

num_rows (int): number of rows to generate start_time (datetime): datetime to start from if timeseries increments (str): if timeseries, you may define the

type of increments you want based on a series of string choices ('days', 'hours', 'seconds', 'random')

end_time (datetime): optional end date if timeseries

see also *parsers.core*

generate()
Generate the dataset (self.records) based on the given schema.

If a timeseries is selected, this method will pass to *Generator.generate_timeseries*

generate_row()
Generate a row based on the parsed schema

NOTE: this uses string matching to determine if the schema is a list, faker definition or one of the matching patterns in *EVAL_REGEX* and then generates data based on those predefined selections.

generate_timeseries()
Generate a timeseries with a *timestamp* column.

This uses the parser start date and increments to

NOTE: a warning will be logged if there is an endtime given and the number of rows is not reached before the endtime. Endtime takes precedence if specified.

TODO: should num_rows take precedence over end time?

increment_time()
For timeseries generation, increment the start time given the parser requirements:

- hours, days, seconds
- if not set, returns random mix

TODO: allow for set interval?

output_filename
Return filename if output follows proper file format file://{[}absolute or relative filepath]

to_output()
Return or create output based on parsed schema.

For more use cases, please reference the *Usage* section.

7.1.8 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:

Types of Contributions

Report Bugs

Report bugs at https://github.com/kjam/datafuzz/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.

Fix Bugs

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

Implement Features

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

Write Documentation

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

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/kjam/datafuzz/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 :)

Get Started!

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

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

```
$ git clone git@github.com:your_name_here/datafuzz.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 datafuzz
$ cd datafuzz/
$ 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 pylint and the tests, including testing other Python versions with tox:

```
$ pylint datafuzz
$ pytest tests --cov=.
$ tox
```

   Our goal is to keep coverage as close to 100% as possible and to keep *pylint* code rating >= 9.5. For *pylint* you will need to set up or edit your *.pylintrc* with the following additional sections:

```
[TYPECHECK]
ignored-modules=numpy,numpy.random,settings
ignored-classes=SQLObject,numpy,numpy.random
```

   For more information, consult *getting started with PyLint <https://docs.pylint.org/en/latest/tutorial.html>* and *Running Pylint <https://pylint.readthedocs.io/en/latest/user_guide/run.html>*.

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.

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 and documentation.
3. The pull request should work for Python 3.4, 3.5 and 3.6. Check https://travis-ci.org/kjam/datafuzz/pull_requests and make sure that the tests pass for all supported Python versions.

Tips

To run a subset of tests:

```
$ pytest --cov=datafuzz.dataset tests/test_dataset.py
```

7.1.9 Credits

Development Lead

- Katharine Jarmul <katharine@kjamistan.com>

Contributors

None yet. Why not be the first?

7.1.10 History

0.1.2 (2023-03-28)

- Update python versions
- Fix several nondeterministic bugs

0.1.0 (2017-09-13)

- First release on PyPI.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index