SilviMetric

Release 1.1.1

Kyle Mann, Howard Butler, Bob McGaughey

Apr 23, 2024

CONTENTS

SilviMetric is an open source library and set of utilities from Hobu, Inc. that are useful for summarizing point cloud data into raster and raster-like products.

Find out more about SilviMetric by visiting *About*. A slide deck about SilviMetric is also available on Google Slides.

CHAPTER

ONE

ABOUT

Summarizing and filtering point cloud data into useful information for modeling is challenging. In forestry applications in particular, the FUSION software toolkit is often used to extract information in preparation for modeling. FUSION, however, has a few missing features that make it

Working with Bob McGaughey and the USFS GTAC team, Kyle Mann and Howard Butler from Hobu, Inc., developed the initial prototype of SilviMetric to implement an alternative approach to computing the "GridMetrics" component of typical FUSION processing pipelines.

SilviMetric does this by breaking apart the computation of metrics into three distinct steps – *info*, *shatter*, and *extract*. SilviMetric takes an infrastructure computing approach to the challenge by applying emerging open source technologies that speak cloud, are nimble with data formats, and compute in a more friendly language - Python.

1.1 Technologies

SilviMetric stands on the shoulders of giants to provide an integrated solution to computing rasterized point cloud metrics. These technologies include:

- PDAL reads point cloud content and allows users to filter or process data as it ingested.
- Dask processes tasks for :ref`shatter` and *extract* in a highly parallel, cloud-friendly distributed computing environment.
- TileDB stores metrics in cloud object stores such as S3 in addition to typical filesystems.
- Python computes metrics and provides a diverse and convenient computing capability for users to easily add and extract their own metrics to the database.

CHAPTER

TWO

COMMAND LINE INTERFACE

```
Usage: silvimetric [OPTIONS] COMMAND [ARGS]...

Options:
-d, --database PATH             Database path
--debug                         Changes logging level from INFO to DEBUG.
--log-dir TEXT                  Directory for log output
--progress BOOLEAN              Report progress
--workers INTEGER               Number of workers for Dask
--threads INTEGER               Number of threads per worker for Dask
--watch                         Open dask diagnostic page in default web
                                browser.
--dasktype [threads|processes]  What Dask uses for parallelization. For
                                moreinformation see here https://docs.dask.o
                                rg/en/stable/scheduling.html#local-threads
--scheduler [distributed|local|single-threaded]
                                Type of dask scheduler. Both are local, but
                                are run with different dask libraries. See
                                more here https://docs.dask.org/en/stable/sc
                                heduling.html.
--help                          Show this message and exit.

Commands:
extract     Extract silvimetric metrics from DATABASE
info        Retrieve information on current state of DATABASE
initialize  Create an empty DATABASE
scan        Scan point cloud and determine the optimal tile size.
shatter     Shatter point clouds into DATABASE cells.
```

2.1 extract

2.1.1 Synopsis

```
Usage: silvimetric [OPTIONS] extract [OPTIONS]

Extract silvimetric metrics from DATABASE

Options:
```

(continues on next page)

(continued from previous page)

```
-a, --attributes ATTRS  List of attributes to include output
-m, --metrics METRICS   List of metrics to include in output
--bounds BOUNDS         Bounds for data to include in output
-o, --outdir PATH       Output directory.  [required]
--help                  Show this message and exit.
```

2.1.2 Example

```
silvimetric -d test.tdb extract -o test_tifs/
```

2.2 shatter

2.2.1 Synopsis

```
Usage: silvimetric [OPTIONS] shatter [OPTIONS] POINTCLOUD

Insert data provided by POINTCLOUD into the silvimetric DATABASE

Options:
--bounds BOUNDS                 Bounds for data to include in processing
--tilesize INTEGER              Number of cells to include per tile
--report                        Whether or not to write a report of the
                                process, useful for debugging
--date [%Y-%m-%d|%Y-%m-%dT%H:%M:%SZ]
                                Date the data was produced.
--dates <DATETIME DATETIME>...  Date range the data was produced during
--help                          Show this message and exit.
```

2.2.2 Example

```
silvimetric -d test.tdb shatter --date 2023-1-1 tests/data/test_data.copc.laz
```

2.3 info

2.3.1 Synopsis

```
Usage: silvimetric [OPTIONS] info [OPTIONS]

Print info about Silvimetric database

Options:
    --bounds BOUNDS                 Bounds to filter by
    --date [%Y-%m-%d|%Y-%m-%dT%H:%M:%SZ]
```

(continues on next page)

(continued from previous page)

```
                                  Select processes with this date
  --history                       Show the history section of the output.
  --metadata                      Show the metadata section of the output.
  --attributes                    Show the attributes section of the output.
  --dates <DATETIME DATETIME>...  Select processes within this date range
  --name TEXT                     Select processes with this name
  --help                          Show this message and exit.
```

2.3.2 Example

```
silvimetric -d test.tdb info
```

2.4 initialize

The *initialize* subcommand constructs the basic TileDB instance to host the SilviMetric data. It can be either a local filesystem path or a S3 URI (eg. `s3://silvimetric/mydata`).

2.4.1 Synopsis

```
Usage: silvimetric [OPTIONS] initialize [OPTIONS]

Initialize silvimetrics DATABASE

Options:
--bounds BOUNDS         Root bounds that encapsulates all data  [required]
--crs CRS               Coordinate system of data  [required]
-a, --attributes ATTRS  List of attributes to include in Database
-m, --metrics METRICS   List of metrics to include in Database
--resolution FLOAT      Summary pixel resolution
--help                  Show this message and exit.
```

2.4.2 Example

```
silvimetric --database test.tdb initialize --crs "EPSG:3857" \
    --bounds '[300, 300, 600, 600]'
```

2.5 scan

2.5.1 Synopsis

```
silvimetric [OPTIONS]  scan [OPTIONS] POINTCLOUD

Options:
--resolution FLOAT     Summary pixel resolution
--filter               Remove empty space in computation. Will take extra
                        time.
--point_count INTEGER  Point count threshold.
--depth INTEGER        Quadtree depth threshold.
--bounds BOUNDS        Bounds to scan.
--help                 Show this message and exit.
```

2.5.2 Usage

```
silvimetric --database test.tdb scan tests/data/test_data.copc.laz
```

CHAPTER

THREE

API

3.1 Resources

3.1.1 Config

class silvimetric.resources.config.**ApplicationConfig**(*debug: bool = (False,)*, *progress: bool = (False,)*, *dasktype: str = 'processes'*, *scheduler: str = 'distributed'*, *workers: int = 12*, *threads: int = 4*, *watch: bool = False*, ***, *tdb_dir: str*, *log: ~silvimetric.resources.log.Log = <factory>*)

Base application config

dasktype: str = 'processes'

Dask parallelization type. For information see https://docs.dask.org/en/stable/scheduling.html#local-threads

debug: bool = (False,)

Debug mode, defaults to False

progress: bool = (False,)

Should processes display progress bars, defaults to False

scheduler: str = 'distributed'

Dask scheduler, defaults to 'distributed'

threads: int = 4

Number of threads per dask worker

watch: bool = False

Open dask diagnostic page in default web browser

workers: int = 12

Number of dask workers

class silvimetric.resources.config.**Config**(***, *tdb_dir: str*, *log: ~silvimetric.resources.log.Log = <factory>*, *debug: bool = False*)

Base config

debug: bool = False

Debug flag.

log: *Log*

Log object.

tdb_dir: str

Path to TileDB directory to use.

class silvimetric.resources.config.**ExtractConfig**(*out_dir: str*, *attrs: list[~silvimetric.resources.entry.Attribute] = <factory>*, *metrics: list[~silvimetric.resources.metric.Metric] = <factory>*, *bounds: ~silvimetric.resources.bounds.Bounds = None*, ***, *tdb_dir: str*, *log: ~silvimetric.resources.log.Log = <factory>*, *debug: bool = False*)

Config for the Extract process.

attrs: list[*Attribute*]

List of attributes to use in shatter. If this is not set it will be filled by the attributes in the database instance.

bounds: *Bounds* = None

The bounding box of the shatter process., defaults to None

metrics: list[*Metric*]

A list of metrics to use in shatter. If this is not set it will be filled by the metrics in the database instance.

out_dir: str

The directory where derived rasters should be written.

class silvimetric.resources.config.**ShatterConfig**(*filename: str, date: ~datetime.datetime | ~typing.Tuple[~datetime.datetime, ~datetime.datetime], attrs: list[~silvimetric.resources.entry.Attribute] = <factory>, metrics: list[~silvimetric.resources.metric.Metric] = <factory>, bounds: ~silvimetric.resources.bounds.Bounds | None = None, name: ~uuid.UUID = UUID('70f56c47-16d5-4ac4-b0f6-6394f0c36db9'), tile_size: int | None = None, start_time: float = 0, end_time: float = 0, point_count: int = 0, mbr: tuple[tuple[tuple[int, int], tuple[int, int]], ...] = <factory>, finished: bool = False, time_slot: int = 0, *, tdb_dir: str, log: ~silvimetric.resources.log.Log = <factory>, debug: bool = False*)

Config for Shatter process

attrs: list[*Attribute*]

List of attributes to use in shatter. If this is not set it will be filled by the attributes in the database instance.

bounds: *Bounds* | None = None

The bounding box of the shatter process., defaults to None

date: datetime | Tuple[datetime, datetime]

A date or date range representing data collection times.

end_time: float = 0

The process ending time in seconds since Jan 1 1970., defaults to 0

filename: str

Input filename referencing a PDAL pipeline or point cloud file.

finished: bool = False

Finished flag for shatter process., defaults to False

mbr: tuple[tuple[tuple[int, int], tuple[int, int]], ...]

The minimum bounding rectangle derived from TileDB array fragments. This will be used to for resuming shatter processes and making sure it doesn't repeat work., defaults to tuple()

metrics: list[*Metric*]

A list of metrics to use in shatter. If this is not set it will be filled by the metrics in the database instance.

name: UUID = UUID('70f56c47-16d5-4ac4-b0f6-6394f0c36db9')

UUID representing this shatter process and will be generated if not provided., defaults to uuid.uuid()

point_count: int = 0

The number of points that has been processed so far., defaults to 0

start_time: float = 0

The process starting time in seconds since Jan 1 1970., defaults to 0

tile_size: int | None = None

The number of cells to include in a tile., defaults to None

time_slot: int = 0

The time slot that has been reserved for this shatter process. Will be used as the timestamp in tiledb writes to better organize and manage processes., defaults to 0

class silvimetric.resources.config.**SilviMetricJSONEncoder**(*, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)

default(*o*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

class silvimetric.resources.config.**StorageConfig**(*root: ~silvimetric.resources.bounds.Bounds*, *crs: pyproj.CRS*, *resolution: float = 30.0*, *attrs: list[~silvimetric.resources.entry.Attribute] = <factory>*, *metrics: list[~silvimetric.resources.metric.Metric] = <factory>*, *version: str = '1.1.1'*, *capacity: int = 1000000*, *next_time_slot: int = 1*, ***, *tdb_dir: str*, *log: ~silvimetric.resources.log.Log = <factory>*, *debug: bool = False*)

Config for constructing a Storage object

attrs: list[*Attribute*]

List of *silvimetric.resources.entry.Attribute* attributes that represent point data, defaults to Z, NumberOfReturns, ReturnNumber, Intensity

capacity: int = 1000000

TileDB Capacity, defaults to 1000000

crs: pyproj.CRS

Coordinate reference system, same for all data in a project

metrics: list[*Metric*]

List of *silvimetric.resources.metric.Metric* Metrics that represent derived data, defaults to values in Metrics object

next_time_slot: int = 1

Next time slot to be allocated to a shatter process. Increment after use., defaults to 1

resolution: float = 30.0

Resolution of cells, same for all data in a project, defaults to 30.0

root: *Bounds*

Root project bounding box

version: str = '1.1.1'

Silvimetric version

3.1.2 Bounds

class silvimetric.resources.bounds.**Bounds**(*minx: float*, *miny: float*, *maxx: float*, *maxy: float*)

Simple class to represent a 2 or 3-dimensional bounding box that can be generated from both JSON or PDAL bounds form.

bisect()

Bisects the current Bounds

Yield
4 child bounds

disjoint(*other*)

Determine if two bounds are disjointed

Parameters
other – Bounds this object is being compared to

Returns
True if this box shares no point with the other box, otherwise False

static from_string(*bbox_str: str*) → Self

Create Bounds object from a PDAL bounds string in the form:

"([1,101],[2,102],[3,103])" "{"minx": 1,"miny": 2,"maxx": 101,"maxy": 102}" "[1,2,101,102]" "[1,2,3,101,102,103]"

Parameters
bbox_str – Bounds string

Raises

- **Exception** – Unable to load Bounds via json or PDAL bounds type
- **Exception** – Bounding boxes must have either 4 or 6 elements

Returns
Bounds object

get() → list[float]

Return Bounds as a list of floats

Returns
list of floats in form [minx, miny, maxx, maxy]

static shared_bounds(*first: Self*, *second: Self*) → Self | None

Find the Bounds that is shared between two Bounds.

Parameters

- **first** – First Bounds object for comparison.
- **second** – Second Bounds object for comparison.

Returns
None if there is no overlap, otherwise the shared Bounds

to_json() → list[float]

Return object as a json serializable list

Returns
list of floats in form [minx, miny, maxx, maxy]

to_string() → str

Return string representation of Bounds

Returns
string of a list of floats in form [minx, miny, maxx, maxy]

maxx

maximum X plane

maxy

maximum Y plane

minx

minimum X Plane

miny

minimum Y plane

3.1.3 Data

class silvimetric.resources.data.**Data**(*filename: str*, *storageconfig:* StorageConfig, *bounds:* Bounds = *None*)

Bases: object

Represents a point cloud or PDAL pipeline, and performs essential operations necessary to understand and execute a Shatter process.

count(*bounds:* Bounds) → int

For the provided bounds, read and count the number of points that are inside them for this instance.

Parameters
bounds – query bounding box

Returns
point count

estimate_count(*bounds:* Bounds) → int

For the provided bounds, estimate the maximum number of points that could be inside them for this instance.

Parameters
bounds – query bounding box

Returns
estimated point count

execute()

Execute PDAL pipeline

Raises
Exception – PDAL error message passed from execution

get_array() → ndarray

Fetch the array from the execute()'d pipeline

Returns
get data as a numpy ndarray

static get_bounds(*reader: pdal.Reader*) → *Bounds*

Get the bounding box of a point cloud from PDAL.

Parameters
reader – PDAL Reader representing input data

Returns
bounding box of point cloud

get_pipeline() → pdal.Pipeline

Fetch the pipeline for the instance

Raises

- **Exception** – File type isn't COPC or EPT
- **Exception** – More than one reader detected

Returns
Return PDAL pipline

get_reader() → pdal.Reader

Grab or make the reader for this instance so we can use it to do things like get the count()

Returns
get PDAL reader for input

is_pipeline() → bool

Does this instance represent a pdal.Pipeline or a simple filename

Returns
Return true if input is a pipeline

make_pipeline() → pdal.Pipeline

Take a COPC or EPT endpoint and generate a PDAL pipeline for it

Returns
Return PDAL pipeline

property array: ndarray

Fetch the array from the execute()'d pipeline

Returns
get data as a numpy ndarray

bounds

Bounds of this section of data

filename

Path to either PDAL pipeline or point cloud file

pipeline

PDAL pipeline

reader

PDAL reader

reader_thread_count

Thread count for PDAL reader. Keep to 2 so we don't hog threads

storageconfig

silvimetric.resources.StorageConfig

3.1.4 Log

log.py Project: CRREL-NEGGS University of Houston Collaboration Date: February 2021

A module for setting up logging.

class silvimetric.resources.log.**Log**(*log_level: int*, *logdir: str = None*, *logtype: str = 'stream'*, *logfilename: str = 'silvimetric-log.txt'*)

Bases: object

debug(*msg: str*)

Forward debug messages down to logger

info(*msg: str*)

Forward info messages down to logger

to_json()

warning(*msg: str*)

Forward warning messages down to logger

3.1.5 Storage

class silvimetric.resources.storage.**Storage**(*config:* StorageConfig, *ctx: Ctx = None*)

Handles storage of shattered data in a TileDB Database.

consolidate_shatter(*proc_num: int*) → None

Consolidate the fragments from a shatter process into one fragment. This makes the database perform better, but reduces the granularity of time traveling.

Parameters

proc_num – Time slot associated with shatter process.

static create(*config:* StorageConfig, *ctx: Ctx = None*)

Creates TileDB storage.

Parameters

config

[StorageConfig] Storage StorageConfig

ctx

[tiledb.Ctx, optional] TileDB Context, by default is None

Returns

Storage

Returns newly created Storage class

Raises

Exception

Raises bounding box errors if not of lengths 4 or 6

delete(*proc_num: int*) → *ShatterConfig*

Delete Shatter process and all associated data from database.

Parameters

proc_num – Shatter process time slot

Returns

Config of deleted Shatter process

static from_db(*tdb_dir: str*)

Create Storage object from information stored in a database.

Parameters

tdb_dir – TileDB database directory.

Returns
Returns the derived storage.

`getAttributes`() → list[*Attribute*]

Find list of attribute names from storage config.

Returns
List of attribute names.

`getConfig`() → *StorageConfig*

Get the StorageConfig currently in use by the Storage.

Returns
StorageConfig representing this object.

`getMetadata`(*key: str*, *timestamp: int*) → str

Return metadata at given key.

Parameters

- **key** – Key to look for in metadata.
- **timestamp** – Time stamp for querying database.

Returns
Metadata value found in storage.

`getMetrics`() → list[*Metric*]

Find List of metric names from storage config

Returns
List of metric names.

`get_fragments_by_time`(*proc_num: int*) → list[FragmentInfo]

Get TileDB array fragments from the time slot specified.

Parameters
proc_num – Requested time slot.

Returns
Array fragments from time slot.

`get_history`(*start_time: datetime*, *end_time: datetime*, *bounds:* Bounds, *name: str = None*)

Retrieve history of the database at current point in time.

Parameters

- **start_time** – Query parameter, starting datetime of process.
- **end_time** – Query parameter, ending datetime of process.
- **bounds** – Query parameter, bounds to query by.
- **name** – Query paramter, shatter process uuid., by default None

Returns
Returns list of array fragments that meet query parameters.

`mbrs`(*proc_num: int*)

Get minimum bounding rectangle of a given shatter process. If this process has been finished and consolidated the mbr will be much less granulated than if the fragments are still intact. Mbrs are represented as tuples in the form of ((minx, maxx), (miny, maxy))

Parameters
proc_num – Process number or time slot of the shatter process.

Returns
Returns mbrs that match the given process number.

open(*mode: str = 'r'*, *timestamp=None*) → SparseArrayImpl

Open stream for TileDB database in given mode and at given timestamp.

Parameters

- **mode** – Mode to open TileDB stream in. Valid options are 'w', 'r', 'm', 'd'., defaults to 'r'.
- **timestamp** – Timestamp to open database at., defaults to None.

Raises

- **Exception** – Incorrect Mode was given, only valid modes are 'w' and 'r'.
- **Exception** – Path exists and is not a TileDB array.
- **Exception** – Path does not exist.

Yield
TileDB array context manager.

reserve_time_slot() → int

Increment time slot in database and reserve that spot for a new shatter process.

Parameters
config – Shatter config will be written as metadata to reserve time slot.

Returns
Time slot.

saveConfig() → None

Save StorageConfig to the Database

saveMetadata(*key: str*, *data: str*, *timestamp: int*) → None

Save metadata to storage.

Parameters

- **key** – Metadata key to save to.
- **data** – Data to save to metadata.

3.1.6 Entry

class silvimetric.resources.entry.**Entry**

Base class for Attribute and Metric. These represent entries into the database.

3.1.7 Attribute

class silvimetric.resources.entry.**Attribute**(*name: str*, *dtype: dtype*, *deps: list[Self] = None*)

Represents point data from a PDAL execution that has been binned, and provides the information necessary to transfer that data to the database.

3.1.8 Metric

class silvimetric.resources.metric.**Metric**(*name: str*, *dtype: dtype*, *method: Callable[[ndarray, Any | None], ndarray]*, *dependencies: list[*Attribute*] = None*)

A Metric is an Entry representing derived cell data. There is a base set of metrics available through Silvimetric, or you can create your own. A Metric object has all the information necessary to facilitate the derivation of data as well as its insertion into the database.

3.1.9 Extents

class silvimetric.resources.extents.**Extents**(*bounds:* Bounds, *resolution: float*, *root:* Bounds)

Handles bounds operations for point cloud data.

chunk(*data:* Data, *res_threshold=100*, *pc_threshold=600000*, *depth_threshold=6*)

Split up a dataset into tiles based on the given thresholds. Unlike Scan this will filter out any tiles that contain no points.

Parameters

- **data** – Incoming Data object to oeprate on.
- **res_threshold** – Resolution threshold., defaults to 100
- **pc_threshold** – Point count threshold., defaults to 600000
- **depth_threshold** – Tree depth threshold., defaults to 6

Returns
Return list of Extents that fit the criteria

disjoint(*other: Self*)

Determined if this Extents shares any points with another Extents object.

Parameters
other – Extents object to compare against.

Returns
True if no shared points, false otherwise.

disjoint_by_mbr(*mbr*)

Determine if this Extents shares any points with a minimum bounding rectangle.

Parameters
mbr – Minimum bounding rectangle as defined by TileDB.

Returns
True if no shared points, false otherwise.

`filter()`

Creates quad tree of chunks for this bounds, runs pdal quickinfo over this to determine if there are any points available. Uses a bottom resolution of 1km.

Parameters

- **data** – Data object containing point cloud details.
- **res_threshold** – Resolution threshold., defaults to 100
- **pc_threshold** – Point count threshold., defaults to 600000
- **depth_threshold** – Tree depth threshold., defaults to 6
- **depth** – Current tree depth., defaults to 0

Returns

Returns a list of Extents.

`static from_storage(tdb_dir: str)`

Create Extents from information stored in database.

Parameters

tdb_dir – TileDB database directory.

Returns

Returns resulting Extents.

`static from_sub(tdb_dir: str, sub: Bounds)`

Create an Extents that is less than the overall extents of the database.

Parameters

- **tdb_dir** – TileDB database directory.
- **sub** – Desired bounding box.

Returns

Returns resulting Extents.

`get_indices()`

Create indices for this section of the database relative to the root bounds.

Returns

Indices of this bounding box

`get_leaf_children(tile_size)`

Get children Extents with given number of cells per tile.

Parameters

tile_size – Cells per tile.

Yield

Yield from list of child extents.

`split()`

Split this extent into 4 children along the cell lines

Returns

Returns 4 child extents

`bounds`

Bounding box of this section of data.

cell_count
Number of cells in this Extents

domain: tuple[tuple[float, float], tuple[float, float]]
Minimum bounding rectangle of this Extents

rangex
Range of X Indices

rangey
Range of Y indices

resolution
Resolution of database.

root
Root bounding box of the database.

x1
Minimum X index

x2
Maximum X index

y1
Minimum Y index

y2
Maximum Y index

3.2 Commands

3.2.1 Initialize

silvimetric.commands.initialize.**initialize**(*storage:* StorageConfig)
Initialize a Silvimetric TileDB instance for a given StorageConfig instance.

Parameters
storage – *silvimetric.resources.config.StorageConfig*.

Returns
silvimetric.resources.storage.Storage database object.

3.2.2 Scan

silvimetric.commands.scan.**extent_handle**(*extent:* Extents, *data:* Data, *res_threshold: int = 100*, *pc_threshold: int = 600000*, *depth_threshold: int = 6*, *log:* Log *= None*) → list[int]
Recurisvely iterate through quad tree of this Extents object with given threshold parameters.

Parameters

- **extent** – Current Extent.
- **data** – Data object created from point cloud file.

- **res_threshold** – Resolution threshold., defaults to 100
- **pc_threshold** – Point count threshold., defaults to 600000
- **depth_threshold** – Tree depth threshold., defaults to 6

Returns
Returns list of Extents that fit thresholds.

silvimetric.commands.scan.**scan**(*tdb_dir: str*, *pointcloud: str*, *bounds:* Bounds, *point_count: int = 600000*, *resolution: float = 100*, *depth: int = 6*, *filter: bool = False*, *log:* Log = *None*)

Scan pointcloud and determine appropriate tile sizes.

Parameters

- **tdb_dir** – TileDB database directory.
- **pointcloud** – Path to point cloud.
- **bounds** – Bounding box to filter by.
- **point_count** – Point count threshold., defaults to 600000
- **resolution** – Resolution threshold., defaults to 100
- **depth** – Tree depth threshold., defaults to 6
- **filter** – Remove empty Extents. This takes longer, but is more accurage., defaults to False

Returns
Returns list of point counts.

3.2.3 Shatter

silvimetric.commands.shatter.**arrange**(*data: tuple[ndarray, ndarray, ndarray]*, *leaf:* Extents, *attrs: list[str]*) → tuple[ndarray, ndarray, dict] | None

Arrange data to fit key-value TileDB input format.

Parameters

- **data** – Tuple of indices and point data array (xis, yis, data).
- **leaf** – `silvimetric.resources.extents.Extent` being operated on.
- **attrs** – List of attribute names.

Raises
Exception – Missing attribute error.

Returns
None if no work is done, or a tuple of indices and rearranged data.

silvimetric.commands.shatter.**cell_indices**(*xpoints*, *ypoints*, *x*, *y*)

Return view of point data that fits in cell (x,y)

silvimetric.commands.shatter.**get_atts**(*points: ndarray*, *leaf:* Extents, *attrs: list[str]*) → list[ndarray[Any, dtype]]

Filter point data to just attributes we want

Parameters

- **points** – Point data.

- **leaf** – *silvimetric.resources.extents.Extents* being operated on.
- **attrs** – List of attribute names.

Returns
Point data filtered to just the desired attributes.

silvimetric.commands.shatter.**get_data**(*extents:* Extents, *filename: str*, *storage:* Storage) → ndarray

Execute pipeline and retrieve point cloud data for this extent

Parameters

- **extents** – *silvimetric.resources.extents.Extents* being operated on.
- **filename** – Path to either PDAL pipeline or point cloud.
- **storage** – *silvimetric.resources.storage.Storage* database object.

Returns
Point data array from PDAL.

silvimetric.commands.shatter.**get_metrics**(*data_in: tuple[ndarray, ndarray, dict]*, *attrs: list[str]*, *storage:* Storage) → tuple[ndarray, ndarray, dict]

Performs metric operations over point data and combine it with the attribute data that is coming in.

Parameters

- **data_in** – Data to run metric methods over.
- **attrs** – List of attributes in the incoming data.
- **storage** – *silvimetric.resources.storage.Storage*.

Returns
Combined point data and metric data.

silvimetric.commands.shatter.**run**(*leaves: Bag*, *config:* ShatterConfig, *storage:* Storage) → int

Coordinate running of shatter process and handle any interruptions

Parameters

- **leaves** – Dask bag of Extent leaf nodes.
- **config** – *silvimetric.resources.config.ShatterConfig*
- **storage** – *silvimetric.resources.storage.Storage*

Returns
Number of points processed.

silvimetric.commands.shatter.**shatter**(*config:* ShatterConfig) → int

Handle setup and running of shatter process. Will look for a config that has already been run before and needs to be resumed.

Parameters
config – *silvimetric.resources.config.ShatterConfig*.

Returns
Number of points processed.

silvimetric.commands.shatter.**write**(*data_in: tuple[ndarray, ndarray, dict]*, *tdb: Array*) → int

Write cell data to database

Parameters

- **data_in** – Data to be written to database.

- **tdb** – TileDB write stream.

Returns
Number of points written.

3.2.4 Info

silvimetric.commands.info.**check_values**(*start_time: datetime*, *end_time: datetime*, *bounds:* Bounds, *name: UUID | str*)

Validate arguments for info command.

Parameters

- **start_time** – Starting datetime object.
- **end_time** – Ending datetime object.
- **bounds** – Bounds to query by.
- **name** – Name to query by.

Raises

- **TypeError** – Incorrect type of start_time argument.
- **TypeError** – Incorrect type of end_time argument.
- **TypeError** – Incorrect type of bounds argument.
- **TypeError** – Incorrect type of name argument.
- **TypeError** – Incorrect type of name argument.

silvimetric.commands.info.**info**(*tdb_dir: str*, *start_time: datetime = None*, *end_time: datetime = None*, *bounds:* Bounds *= None*, *name: str | UUID = None*) → dict

Collect information about database in current state

Parameters

- **tdb_dir** – TileDB database directory path.
- **start_time** – Process starting time query, defaults to None
- **end_time** – Process ending time query, defaults to None
- **bounds** – Bounds query, defaults to None
- **name** – Name query, defaults to None

Returns
Returns json object containing information on database.

3.2.5 Extract

silvimetric.commands.extract.**extract**(*config:* ExtractConfig) → None

Pull data from database for each desired metric and output them to rasters

Parameters
config – ExtractConfig.

silvimetric.commands.extract.**get_metrics**(*data_in: Dict[str, ndarray]*, *config:* ExtractConfig) → Dict[str, ndarray]

Reruns a metric over this cell. Only called if there is overlapping data.

Parameters

- **data_in** – Cell data to be rerun.
- **config** – ExtractConfig.

Returns
Combined dict of attribute and newly derived metric data.

silvimetric.commands.extract.**handle_overlaps**(*config:* ExtractConfig, *storage:* Storage, *indices: ndarray*) → pandas.DataFrame

Handle cells that have overlapping data. We have to re-run metrics over these cells as there's no other accurate way to determined metric values. If there are no overlaps, this will do nothing.

Parameters

- **config** – ExtractConfig.
- **storage** – Database storage object.
- **indices** – Indices with overlap.

Returns
Dataframe of rerun data.

silvimetric.commands.extract.**write_tif**(*xsize: int*, *ysize: int*, *data: ndarray*, *name: str*, *config:* ExtractConfig) → None

Write out a raster with GDAL

Parameters

- **xsize** – Length of X plane.
- **ysize** – Length of Y plane.
- **data** – Data to write to raster.
- **name** – Name of raster to write.
- **config** – ExtractConfig.

3.2.6 Delete

silvimetric.commands.manage.**delete**(*tdb_dir: str*, *name: str*) → *ShatterConfig*

Delete Shatter process from database and return config for that process.

Parameters

- **tdb_dir** – TileDB database directory path.
- **name** – UUID name of the Shatter process.

Raises

- **KeyError** – Shatter process with ID does not exist.
- **ValueError** – Shatter process with ID is missing a time reservation

Returns

Config of process that was deleted.

3.2.7 Resume

silvimetric.commands.manage.**resume**(*tdb_dir: str*, *name: str*) → int

Resume partially completed shatter process. Process must partially completed and have an already established time slot.

Parameters

- **tdb_dir** – TileDB database directory path.
- **name** – UUID name of Shatter process.

Returns

Point count of the restarted shatter process.

3.2.8 Restart

silvimetric.commands.manage.**restart**(*tdb_dir: str*, *name: str*) → int

Delete shatter process from database and run it again with the same config.

Parameters

- **tdb_dir** – TileDB database directory path.
- **name** – UUID name of Shatter process.

Returns

Point count of the restarted shatter process.

CHAPTER

FOUR

QUICKSTART

SilviMetric depends upon Conda for packaging support. You must first install all of SilviMetric's dependencies using Conda:

This tutorial shows you how to *initialize*, *shatter*, and *extract* data in SilviMetric using the *Command Line Interface*.

We are going to use the Autzen Stadium as our test example.

Note: The Autzen Stadium has units in feet, and this can sometimes be a source of confusion for tile settings and such.

4.1 Installation

Open a Conda terminal and install necessary dependencies

```
conda env create \
    -f https://raw.githubusercontent.com/hobuinc/silvimetric/main/
↪environment.yml  \
    -n silvimetric
```

Note: We are installing the list of dependencies as provided by the SilviMetric GitHub listing over the internet.

Warning: If you are using windows, line continuation characters are ^ instead of \

2. Activate the environment:

```
conda activate silvimetric
```

3. Install SilviMetric:

```
pip install silvimetric
```

4.2 Initialization

Initialize a SilviMetric database. To initialize a SilviMetric database, we need a bounds and a coordinate reference system.

1. **We first need to determine a bounds for our database. In our case,**
   we are going to use PDAL and jq to grab our bounds

   ```
   pdal info https://s3.amazonaws.com/hobu-lidar/autzen-classified.copc.laz  \
       --readers.copc.resolution=1 | jq -c '.stats.bbox.native.bbox'
   ```

   Our boundary is emitted in expanded form.

   ```
   {"maxx":639003.73,"maxy":853536.21,"maxz":615.26,"minx":635579.2,"miny":848884.
   ↪83,"minz":406.46}
   ```

   Note: You can express bounds in two additional formats for SilviMetric:

   - `[635579.2, 848884.83, 639003.73, 853536.21]` – `[minx, miny, maxx, maxy]`
   - `([635579.2,848884.83],[639003.73,853536.2])` – `([minx, maxx], [miny, maxy])`

   Note: You can install jq by issuing `conda install jq -y` in your environment if you are on Linux or Mac. On Windows, you will need to download jq from the website and put it in your path. https://jqlang.github.io/jq/download/

2. **We need a coordinate reference system for the database. We will grab it from**
   the PDAL metadata just like we did for the bounds.

   ```
   pdal info --metadata https://s3.amazonaws.com/hobu-lidar/autzen-classified.copc.
   ↪laz  \
       --readers.copc.resolution=10 | \
       jq -c '.metadata.srs.json.components[0].id.code'
   ```

   Our EPSG code is in the `pdal info --metadata` output, and after extracted by jq, we can use it.

   ```
   2992
   ```

Note: Both a bounds and CRS must be set to initialize a database. We can set them to whatever we want, but any data we are inserting into the database must match the coordinate system of the SilviMetric database.

3. With bounds and CRS in hand, we can now initialize the database

   ```
   silvimetric autzen-smdb.tdb \
       initialize \
       '{"maxx":639003.73,"maxy":853536.21,"maxz":615.26,"minx":635579.2,"miny
   ↪":848884.83,"minz":406.46}' \
       EPSG:2992
   ```

Note: Be careful with your shell's quote escaping rules!

4.3 Scan

The *scan* command will tell us information about the pointcloud with respect to the database we already created, including a best guess at the correct number of cells per tile, or *tile size*.

```
silvimetric -d ${db_name} scan ${pointcloud}
```

We should see output like the output below, recommending we use a *tile size* of 185.

```
silvimetric - INFO - info:156 - Pointcloud information:
silvimetric - INFO - info:156 -   Storage Bounds: [635579.2, 848884.83, 639003.73,␣
↪853536.21]
silvimetric - INFO - info:156 -   Pointcloud Bounds: [635577.79, 848882.15, 639003.73,␣
↪853537.66]
silvimetric - INFO - info:156 -   Point Count: 10653336
silvimetric - INFO - info:156 - Tiling information:
silvimetric - INFO - info:156 -   Mean tile size: 91.51758793969849
silvimetric - INFO - info:156 -   Std deviation: 94.31396536316173
silvimetric - INFO - info:156 -   Recommended split size: 185
```

4.4 Shatter

We can now insert data into the SMDB.

If we run this command without the argument *–tilesize*, *Silvimetric* will determine a tile size for you. The method will be the same as the *Scan* method, but will filter out the tiles that have no data in them.

```
silvimetric -d autzen-smdb.tdb \
   --threads 4 \
   --workers 4 \
   --watch \
   shatter \
   --date 2008-12-01 \
   https://s3.amazonaws.com/hobu-lidar/autzen-classified.copc.laz
```

If we grab the tile size from the *scan* that we ran earlier, we'll skip the filtering step.

```
silvimetric -d autzen-smdb.tdb \
   --threads 4 \
   --workers 4 \
   --watch \
   shatter \
   --tilesize 185 \
   --date 2008-12-01 \
   https://s3.amazonaws.com/hobu-lidar/autzen-classified.copc.laz
```

4.5 Extract

After data is inserted, we can extract it into different rasters. When we created the database we gave it a list of *Attributes* and *Metrics*. When we ran *Shatter*, we filled in the values for those in each cell. If we have a database with the *Attributes* Intensity and Z, in combination with the *Metrics* min and max, each cell will contain values for *min_Intensity*, *max_Intensity*, *min_Z*, and *max_Z*. This is also the list of available rasters we can extract.

```
silvimetric -d autzen-smdb.tdb extract -o output-directory
```

4.6 Info

We can query past shatter processes and the schema for the database with the Info call.

```
silvimetric -d autzen-smdb.tdb info --history
```

This will print out a JSON object containing information about the current state of the database. We can find the *name* key here, which necessary for *Delete*, *Restart*, and *Resume*. For the following commands we will have copied the value of the *name* key in the variable *uuid*.

4.7 Delete

We can also remove a *shatter* process by using the *delete* command. This will remove all data associated with that shatter process from the database, but will leave an updated config of it in the database config should you want to reference it later.

```
silvimetric -d autzen-smdb.tdb delete --id $uuid
```

4.8 Restart

If you would like to rerun a *Shatter* process, whether or not it was previously finished, you can use the *restart* command. This will call the *delete* method and use the config from that to re-run the *shatter* process.

```
silvimetric -d autzen-smdb.tdb restart --id $uuid
```

4.9 Resume

If a *Shatter* process is cancelled partway through, we can pick up where we left off with the *Resume* command.

```
silvimetric -d autzen-smdb.tdb resume --id $uuid
```

CHAPTER

FIVE

DEVELOPMENT

SilviMetric is released under the Apache 2.0 License.

SilviMetric is managed and developed on GitHub at https://github.com/hobuinc/silvimetric

CHAPTER

SIX

TUTORIAL

Author
Kyle Mann

Contact
kyle@hobu.co

Date
2/05/2024

Table of Contents

This tutorial will cover how to interact with SilviMetric, including the key commands *initialize*, *info*, *scan*, *shatter*, and *extract*. These commands make up the core functionality of SilviMetric and will allow you convert point cloud files into a storage system with TileDB and extract the metrics that are produced into rasters or read with the library/language of your choice.

6.1 Introduction

SilviMetric was created as a spiritual successor to FUSION, a C++ library written by Robert McGaughey, with a focus on the point cloud metric extraction and management capability that FUSION provides in the form of GridMetrics. SilviMetric aims to handle the challenge of computing statistics and metrics from LiDAR data by using Python instead of C++, delegate data management to TileDB, and leverage the wealth of capabilities provided in the machine learning and scientific computing ecosystem of Python. The goal is to create a library and command line utilities that a wider audience of researchers and developers can contribute to, support distributed computing and storage systems in the cloud, and provide a convenient solution to the challenge of distributing and managing LiDAR metrics that are typically used for forestry modeling.

6.2 Technologies

6.2.1 TileDB

TileDB is an open source database engine for array data. Some features that it provides that make it especially attractive for the challenge that SilviMetric has include:

- Sophisticated "infinite" sparse array support
- Time travel
- Multiple APIs (Python, C++, R, Go, Java)
- Cloud object store (S3, AzB, GCS)

6.2.2 PDAL

PDAL provides point cloud processing, translation, and data conditioning utilities that SilviMetric depends upon to read, ingest, and process point cloud data.

6.2.3 Dask

Dask provides the parallel execution engine for SilviMetric.

6.2.4 NumPy

NumPy is the array processing library of Python that other libraries in the ecosystem build upon including PDAL, SciPy, scikit-learn and PyTorch/Tensorflow.

6.3 Command Line Interface Usage

6.3.1 Base

The base options for SilviMetric include setup options that include dask setup options, log setup options, and progress reporting options. The click python library requires that commands and options associated with specific groups appear in certain orders, so our base options will always be first.

```
Usage: silvimetric [OPTIONS] COMMAND [ARGS]...

Options:
-d, --database PATH             Database path
--debug                         Print debug messages?
--log-level TEXT                Log level (INFO/DEBUG)
--log-dir TEXT                  Directory for log output
--progress BOOLEAN              Report progress
--workers INTEGER               Number of workers for Dask
--threads INTEGER               Number of threads per worker for Dask
--watch                         Open dask diagnostic page in default web
                                browser.
--dasktype [threads|processes]  What Dask uses for parallelization. For
```

(continues on next page)

(continued from previous page)

```
                                moreinformation see here https://docs.dask.o
                                rg/en/stable/scheduling.html#local-threads
--scheduler [distributed|local|single-threaded]
                                Type of dask scheduler. Both are local, but
                                are run with different dask libraries. See
                                more here https://docs.dask.org/en/stable/sc
                                heduling.html.
--help                          Show this message and exit.
```

6.3.2 Initialize

initialize will create a TileDB database that will house all future information that is collected about processed point clouds, including attribute data collected about point in a cell, as well as the computed metrics for each individual combination of *Attribute* and *Metric* for each cell.

Here you will need to define the root bounds of the data, which can be larger than just one dataset, as well as the coordinate system it will live in. You will also need to define any *Attributes* and *Metrics*, as these will be propagated to future processes.

Example:

```
$ DB_NAME="western-us.tdb"
$ BOUNDS="[-14100053.268191, 3058230.975702, -11138180.816218, 6368599.176434]"
$ EPSG=3857

$ silvimetric --database $DB_NAME initialize --bounds "$BOUNDS" --crs "EPSG:$EPSG"
```

Usage:

```
Usage: silvimetric initialize [OPTIONS]

Options:
--bounds BOUNDS         Root bounds that encapsulates all data  [required]
--crs CRS               Coordinate system of data  [required]
-a, --attributes ATTRS  List of attributes to include in Database
-m, --metrics METRICS   List of metrics to include in Database
--resolution FLOAT      Summary pixel resolution
--help                  Show this message and exit.
```

6.3.3 User-Defined Metrics

SilviMetric supports creating custom user defined Metrics not provided by the base software. These behave the same as provided Metrics, but can be defined per-database.

You can create a metrics module by following the sample below, and substituting any number of extra metrics in place of `p75` and `p90`. When looking for a metrics module, we look for a method named `metrics`, and that it returns a list of Metric objects. The methods that are included in these objects need to be able to be serialized by dill in order to be pushed and fetched to and from the database.

```
1 import numpy as np
2
```

(continues on next page)

(continued from previous page)

```python
from silvimetric.resources import Metric

def metrics() -> list[Metric]:

    def p75(arr: np.ndarray):
        return np.percentile(arr, 75)
    m_p75 = Metric(name='p75', dtype=np.float32, method = p75)

    def p90(arr: np.ndarray):
        return np.percentile(arr, 90)
    m_p90 = Metric(name='p90', dtype=np.float32, method = p90)

    return [m_p75, m_p90]
```

When including the metrics in the *initialize* step, be sure to include them by doing *-m './path/to/metrics.py'*. At this point, you will need to have all other metrics you would like to include as well.

Example:

```
$ METRIC_PATH="./path/to/python_metrics.py"
$ silvimetric --database $DB_NAME initialize --bounds "$BOUNDS" \
      --crs "EPSG:$EPSG" \
      -m $METRIC_PATH -m min -m max -m mean
```

Warning: Additional Metrics cannot be added to a SilviMetric after it has been initialized at this time.

6.3.4 Scan

scan will allow us to look through the nodes of the point cloud file that you'd like to run in order to determine a good number of cells you should include per tile. The *Shatter* process will take steps to try to inform itself of the best splits possible before doing it's work. The filter process will remove any sections of the bounds that are empty before we get to the shatter process, removing some wasted compute time. By performing a scan ahead of time though, you only need to do it once.

When scan looks through each section of the data, it looks to see how many points are here, how much area this section is taking up, and what depth we're at in the octree. If any of these pass the defined thresholds, then we stop splitting and return that tile. The number of cells in that tile further tells us how best to split the data.

One standard deviation from the mean is a good starting point for a shatter process, but this won't always be perfect for your use case.

Usage:

```
Usage: silvimetric scan [OPTIONS] POINTCLOUD

Scan point cloud and determine the optimal tile size.

Options:
--resolution FLOAT     Summary pixel resolution
--filter               Remove empty space in computation. Will take extra
                        time.
--point_count INTEGER  Point count threshold.
```

(continues on next page)

(continued from previous page)

```
--depth INTEGER        Quadtree depth threshold.
--bounds BOUNDS        Bounds to scan.
--help                 Show this message and exit.
```

Example:

```
$ FILEPATH="https://s3-us-west-2.amazonaws.com/usgs-lidar-public/MT_
↪RavalliGraniteCusterPowder_4_2019/ept.json"
$ silvimetric -d $DB_NAME --watch scan $FILEPATH
```

Output:

```
2024-02-05 17:29:21,464 - silvimetric - INFO - scan:24 - Tiling information:
2024-02-05 17:29:21,465 - silvimetric - INFO - scan:25 -   Mean tile size: 447.
↪98609121670717
2024-02-05 17:29:21,465 - silvimetric - INFO - scan:26 -   Std deviation: 38695.
↪06897023395
2024-02-05 17:29:21,465 - silvimetric - INFO - scan:27 -   Recommended split size: 39143
```

6.3.5 Shatter

shatter is where the vast majority of the processing happens. Here SilviMetric will take all the previously defined variables like the bounds, resolution, and our tile size, and it will split all data values up into their respective bins. From here, SilviMetric will perform each *Metric* previously defined in *initialize* over the data in each cell. At the end of all that, this data will be written to a *SparseArray* in *TileDB*, where it will be much easier to access.

Usage:

```
Usage: silvimetric shatter [OPTIONS] POINTCLOUD

Options:
--bounds BOUNDS                 Bounds for data to include in processing
--tilesize INTEGER              Number of cells to include per tile
--report                        Whether or not to write a report of the
                                process, useful for debugging
--date [%Y-%m-%d|%Y-%m-%dT%H:%M:%SZ]
                                Date the data was produced.
--dates <DATETIME DATETIME>...  Date range the data was produced during
--help                          Show this message and exit.
```

Example:

```
$ BOUNDS='[-12317431.810079003, 5623829.111356639, -12304931.810082098, 5642881.
↪670239899]'
$ silvimetric -d $DB_NAME \
      --watch shatter $FILEPATH \
      --tilesize 100 \
      --date 2024-02-05 \
      --report \
      --bounds $BOUNDS
```

6.3.6 Info

info provides the ability to inspect the SilviMetric database. Here you can see past *Shatter* processes that have been run, including point counts, attributes, metrics, and other process metadata.

Usage:

```
Usage: silvimetric info [OPTIONS]

Options:
--bounds BOUNDS                 Bounds to filter by
--date [%Y-%m-%d|%Y-%m-%dT%H:%M:%SZ]
                                Select processes with this date
--dates <DATETIME DATETIME>...  Select processes within this date range
--name TEXT                     Select processes with this name
--help                          Show this message and exit.
```

Example:

```
$ silvimetric -d $DB_NAME info
```

Output:


```
{
    "attributes": [
        {
            "name": "Z",
            "dtype": "<f8",
            "dependencies": null
        }
    ],
    "metadata": {
        "tdb_dir": "western-us.tdb",
        "log": {
            "logdir": null,
            "log_level": "INFO",
            "logtype": "stream",
            "logfilename": "silvimetric-log.txt"
        },
        "debug": false,
        "root": [
            -14100053.268191,
            3058230.975702,
            -11138180.816218,
            6368599.176434
        ],
        "crs": {"PROJJSON"}
        "resolution": 30.0,
        "attrs": [
            {
                "name": "Z",
                "dtype": "<f8",
                "dependencies": null
            }
```


(continues on next page)

(continued from previous page)

```
        ],
        "metrics": [
            {
                "name": "mean",
                "dtype": "<f4",
                "dependencies": null,
                "method_str": "def m_mean(data):\n    return np.mean(data)\n",
                "method":
↪"gASVKwAAAAAAAACMHHNpbHZpbWV0cmljLnJlc291cmNlcy5tZXRyaWOUjAZtX21lYW6Uk5Qu"
            }
        ],
        "version": "0.0.1",
        "capacity": 1000000
    },
    "history": []
}
```

6.3.7 Extract

extract is the final stop, where SilviMetric outputs the metrics that been binned up nicely, and will output them as rasters to where you select.

Usage:

```
Usage: silvimetric extract [OPTIONS]

Extract silvimetric metrics from DATABASE

Options:
-a, --attributes ATTRS  List of attributes to include output
-m, --metrics METRICS   List of metrics to include in output
--bounds BOUNDS         Bounds for data to include in output
-o, --outdir PATH       Output directory.  [required]
--help                  Show this message and exit.
```

Example:

```
$ OUT_DIR="western-us-tifs"
$ silvimetric -d $DB_NAME extract --outdir $OUT_DIR
```

6.4 Python API Usage

Everything that can be done from the command line can also be performed from within Python. The CLI provides some nice wrapping around some of the setup pieces, including config, log, and Dask handling, but all of these are pieces that you can set up on your own as well.

```
import os
from pathlib import Path
import numpy as np
import pdal
```

(continues on next page)

(continued from previous page)

```
import json
from dask.distributed import Client
import webbrowser

from silvimetric.resources import Storage, Metric, Metrics, Bounds, Pdal_Attributes
from silvimetric.resources import StorageConfig, ShatterConfig, ExtractConfig
from silvimetric.commands import scan, shatter, extract

########## Setup #############

# Here we create a path for our current working directory, as well as the path
# to our forest data, the path to the database directory, and the path to the
# directory that will house the raster data.
curpath = Path(os.path.dirname(os.path.realpath(__file__)))
filename = "https://s3-us-west-2.amazonaws.com/usgs-lidar-public/MT_
↪RavalliGraniteCusterPowder_4_2019/ept.json"
db_dir_path = Path(curpath  / "western_us.tdb")

db_dir = str(db_dir_path)
out_dir = str(curpath / "westsern_us_tifs")
resolution = 10 # 10 meter resolution

# we'll use PDAL python bindings to find the srs of our data, and the bounds
reader = pdal.Reader(filename)
p = reader.pipeline()
qi = p.quickinfo[reader.type]
bounds = Bounds.from_string((json.dumps(qi['bounds'])))
srs = json.dumps(qi['srs']['json'])

######## Create Metric ########
# Metrics give you the ability to define methods you'd like applied to the data
# Here we define, the name, the data type, and what values we derive from it.

def make_metric():
    def p75(arr: np.ndarray):
        return np.percentile(arr, 75)

    return Metric(name='p75', dtype=np.float32, method = p75)

###### Create Storage #####
# This will create a tiledb database, same as the `initialize` command would
# from the command line. Here we'll define the overarching bounds, which may
# extend beyond the current dataset, as well as the CRS of the data, the list
# of attributes that will be used, as well as metrics. The config will be stored
# in the database for future processes to use.

def db():
    perc_75 = make_metric()
    attrs = [
        Pdal_Attributes[a]
        for a in ['Z', 'NumberOfReturns', 'ReturnNumber', 'Intensity']
    ]
```

(continues on next page)

(continued from previous page)

```
    metrics = [
        Metrics[m]
        for m in ['mean', 'min', 'max']
    ]
    metrics.append(perc_75)
    st_config = StorageConfig(db_dir, bounds, resolution, srs, attrs, metrics)
    storage = Storage.create(st_config)

###### Perform Shatter #####
# The shatter process will pull the config from the database that was previously
# made and will populate information like CRS, Resolution, Attributes, and what
# Metrics to perform from there. This will split the data into cells, perform
# the metric method over each cell, and then output that information to TileDB

def sh():
    sh_config = ShatterConfig(db_dir, filename, tile_size=200)
    with Client(n_workers=10, threads_per_worker=3, timeout=100000) as client:
        webbrowser.open(client.cluster.dashboard_link)
        shatter(sh_config, client)


###### Perform Extract #####
# The Extract step will pull data from the database for each metric/attribute combo
# and store it in an array, where it will be output to a raster with the name
# `m_{Attr}_{Metric}.tif`. By default, each computed metric will be written
# to the output directory, but you can limit this by defining which Metric names
# you would like
def ex():
    ex_config = ExtractConfig(db_dir, out_dir)
    extract(ex_config)

####### Perform Scan #######
# The Scan step will perform a search down the resolution tree of the COPC or
# EPT file you've supplied and will provide a best guess of how many cells per
# tile you should use for this dataset.

def sc():
    scan.scan()


if __name__ == "__main__":
    make_metric()
    db()
    sh()
    ex()
```

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

F

G

H

I

L

M

N

O

P

R

S

T

V

W

X

Y