UNICEF Mobility Pipeline

Stanford Code the Change (http://codethechange.stanford.edu)

Mar 16, 2020

CONTENTS:

CHAPTER

ONE

GETTING STARTED

Our code is hosted here: https://github.com/codethechange/mobility_pipeline

1.1 Getting the Code and Dependencies

1. Choose where you want to download the code, and navigate to that directory. Then download the code.

```
$ cd path/to/desired/directory
$ git clone https://github.com/codethechange/mobility_pipeline.git
```

2. Install python 3 from https://python.org or via your favorite package manager
3. Install dependencies into a virtual environment

```
$ cd name_of_cloned_repository
$ python3 -m venv venv
$ source venv/bin/activate
$ pip install -r requirements.txt
```

Now you're all set!

CHAPTER

TWO

CONTRIBUTING

2.1 Your First Contribution

1. Create a fork of this repository on GitHub under your own account.
2. Follow the getting started guide, substituting references to the main repository for your fork.
3. Create a new branch

```
$ git checkout -b my-new-branch
```

4. Make some awesome commits

```
$ # Make some changes
$ git commit
```

5. Make sure all tests pass

```
$ ./test.sh
$ # All tests should pass, and pylint and mypy should raise no complaints
```

6. Merge in any changes from the main repository that might have occurred since you made the fork. Fix any merge conflicts

```
$ git checkout master
$ git pull upstream master
$ git checkout my-new-branch
$ git merge master
```

7. Push the branch:

```
$ git push -u origin my-new-branch
```

8. Submit a pull request on GitHub
9. Thanks for your contribution! One of the maintainers will get back to you soon with any suggested changes or feedback.

2.2 Guidelines

Any code contributions should follow the following guidelines.

2.2.1 Code Style

Python code should conform to the PEP8 style guidelines.

Docstrings should conform to the Google Style. For example (copied from Google's Style Guide):

```
def fetch_bigtable_rows(big_table, keys, other_silly_variable=None):
    """Fetches rows from a Bigtable.

    Retrieves rows pertaining to the given keys from the Table instance
    represented by big_table.  Silly things may happen if
    other_silly_variable is not None.

    Args:
        big_table: An open Bigtable Table instance.
        keys: A sequence of strings representing the key of each table row
            to fetch.
        other_silly_variable: Another optional variable, that has a much
            longer name than the other args, and which does nothing.

    Returns:
        A dict mapping keys to the corresponding table row data
        fetched. Each row is represented as a tuple of strings. For
        example:

        {'Serak': ('Rigel VII', 'Preparer'),
         'Zim': ('Irk', 'Invader'),
         'Lrrr': ('Omicron Persei 8', 'Emperor')}

        If a key from the keys argument is missing from the dictionary,
        then that row was not found in the table.

    Raises:
        IOError: An error occurred accessing the bigtable.Table object.
    """
```

2.2.2 Testing

To run all tests, execute `test.sh`. All tests should pass on your submission.

Travis CI

These tests are checked are run by Travis CI on all pull requests and the master branch. Before each commit, run `test.sh` and ensure that all tests pass. All tests should pass on each commit to make reverting easy.

Unit Testing

Unit testing is performed using pytest. To run these tests, execute `python -m pytest` from the repository root.

Code and Style Analysis

PEP8 are checked by `pylint`. `pylint` also performs static code analysis to catch some programming errors. This analysis is intended to be a fall-back defense, as unit testing should be thorough.

Type Checking

All code should use type hints wherever type cannot be inferred. At a minimum, all function prototypes should have type hints for the return value and each parameter. Type hinting is performed in the code itself, not in docstrings. Static type analysis is performed by `mypy`

Code Coverage

When running the test suite using `test.sh`, code coverage is computed by pytest-cov when running `pytest` and output after test results. Use these results to ensure that all tests are being covered. If the total coverage is not `100%`, run `coverage report -m` to see which lines were not tested. Incomplete coverage may be acceptable if the untested lines should not have been tested (e.g. code stubs for un-implemented functions).

Coverage is tracked by Codecov.

CHAPTER

THREE

USING MOBILITY_PIPELINE

3.1 Setup

3.1.1 Getting Shapefiles

Download the shapefile (`.shp`) from GADM (Database of Global Administrative Areas). After unpacking the zip file, be sure to choose the `.shp` file that corresponds to the level of admin you want (e.g. states versus counties in the United States).

Next, conver the `.shp` file into a GeoJSON format via the command line tool `ogr2ogr`, available by installing GDAL. Then, do the conversion with this command:

```
ogr2ogr -f GeoJSON -t_srs crs:84 [name].geojson [name].shp
```

replacing `[name]` to match your `.shp` file.

3.1.2 Configuration

Regardless of what you want to use mobility_pipeline for, you will need to tell it how to find your data. You can do so by adjusting the constants in *mobility_pipeline.data_interface*. These constants are:

- *mobility_pipeline.data_interface.DATA_PATH*: The path to the folder holding your data.
- *mobility_pipeline.data_interface.TOWERS_PATH*: This is a CSV file containing the coordinates of each cell tower.
- *mobility_pipeline.data_interface.VORONOI_PATH*: This is a JSON file that describes the cells of a Voronoi Tessellation whose seeds are the cell towers.
- *mobility_pipeline.data_interface.MOBILITY_PATH*: This is a CSV file holding the mobility data.
- *mobility_pipeline.data_interface.TOWER_PREFIX*: This is a string that prefixes tower indices in the tower name. For example, Brazil towers might be named `br0`, `br1`, etc. with `br` being the prefix.

For details on the required formats of these files, see the documentation for `data_interface`.

3.2 Running the Program

There are 2 scripts:

- `gen_country_matrices.py`: run once for each admin level / country you want data for. It will generate the admin-to-tower and tower-to-admin matrices.
- `gen_day_mobility.py`: run for each day's worth of data. It will compute the admin-to-admin mobility data.

For both scripts, run with `--help` for more usage information. Both scripts also run independently of the path constants in `data_interface.py`. Instead, they accept command-line arguments that define their operation.

3.3 Running Utilities

This program also comes with utilities. These utilities use the constants in `data_interface.py` for the most part.

3.3.1 Plot Voronoi

To get a sense for what the Voronoi tessellation looks like, you can plot it by running `plot_voronoi.py`. This will display a plot of all the Voronoi cells with the tower positions overlayed. For details, see the documentation for *mobility_pipeline.plot_voronoi*.

3.3.2 Visualize Overlaps

To see what the overlaps look like, you can see a plot of the shapefile with one Voronoi cell and the admin regions it might overlap with color-coded. The script also prints out the values that would go in the tower-to-admin matrix so you can see what the numbers represent visually. To see the plot, run `visualize_overlaps.py`. For details, see *mobility_pipeline.visualize_overlaps*.

3.3.3 Validate Data File Formats

You can run some checks to provide some assurance that a set of data files are formatted as the program expects. After configuring the data paths as described above, you can run these checks by executing `check_validation.py`. You can also look at the code in this file to see what format the program expects. For details, see *mobility_pipeline.check_validation*.

CHAPTER

FOUR

DEVELOPER MANUAL

4.1 Technical Concepts

4.1.1 Matrices

We store program data internally as numpy matrices wherever possible so that we can take advantage of numpy's optimizations. These are the main matrices the data goes through:

- Tower-to-tower matrix: The raw mobility data between towers
- Tower-to-admin matrix: Stores the fraction of each administrative region that is within the range of each tower (covered by each tower's Voronoi cell).
- Admin-to-tower matrix: Stores the fraction of each tower's range (Voronoi cell) that is within each administrative region.
- Admin-to-admin matrix: The end product, which describes mobility between administrative regions.

For details, see *mobility_pipeline.lib.make_matrix*.

4.2 Code Layout and Organization

4.2.1 Utilities

The python files outside of `lib` are executable script files meant to be run from the terminal. While they may have functions, those functions are not meant to be imported into other programs.

4.2.2 Library

The files within `lib` form a library of functions that are format-agnostic and designed to be repurposed in other programs. They are covered by unit tests and are used by the utilities. They are broadly divided into the following files:

- *mobility_pipeline.lib.make_matrix*: Functions for making and working with the matrices.
- *mobility_pipeline.lib.overlap*: Functions for working with polygon overlaps in general.
- *mobility_pipeline.lib.validate*: Functions for validating data formats.
- *mobility_pipeline.lib.voronoi*: Functions for working with Voronoi tessellations.

4.3 General Computation Process

We get the tower-to-tower matrix from the mobility data. Then, we use the country shapefile and Voronoi tessellation to compute the tower-to-admin and admin-to-tower matrices. Finally, we compute the admin-to-admin matrix by multiplying the other three matrices like this: `(tower-to-admin) * (tower-to-tower) * (admin-to-tower)`

CHAPTER

FIVE

MOBILITY_PIPELINE

5.1 mobility_pipeline package

5.1.1 Subpackages

mobility_pipeline.lib package

Submodules

mobility_pipeline.lib.make_matrix module

Functions for making tower-tower, tower-admin, and admin-tower matrices

Matrices:

- tower-tower: The raw mobility data between cell towers. The value at row `i` and column `j` is the number of people who move on that day from the region served by tower `i` to the region served by tower `j`. Note that really, this is the number of cell phones that connect to tower `i` in the morning and tower `j` in the evening, which we assume represents a person moving. This matrix has row indices of the origin towers and column indices of the destination towers.
- tower-admin: Computed from the Voronoi tessellation and the country shapefile, this matrix represents the percent of each admin that is covered by each tower. For any `x` in the matrix at row `i` and column `j`, we know that a fraction `x` of the admin with index `i` is covered by the tower with index `j`. This means that the matrix has row indices of admin indices and column indices of tower indices.
- admin-tower: Computed from the Voronoi tessellation and the country shapefile, this matrix represents the percent of each tower's range that is within each admin. For any `x` in the matrix at row `i` and column `j`, we know that a fraction `x` of the Voronoi cell for the tower with index `i` is within the admin with index `j`. This means that the matrix has row indices of tower indices and column indices of admin indices.
- admin-admin: This is the final mobility matrix, which represents the number of people who move between admins each day. The value at row `i` and column `j` is the number of people who move on that day from the admin with index `i` to the admin with index `j`. This is, of course, being estimated from cell phone data and the overlaps as computed in the other matrices.

This strategy is explained by Mike Fabrikant at UNICEF: https://medium.com/@mikefabrikant/cell-towers-chiefdoms-and-anonymized-call-detail-records-a-guide-to-creating-a-mobility-matrix-d2d5c1bafb68

`mobility_pipeline.lib.make_matrix.`**`generate_rtree`**(*polygons: collections.abc.Sequence*) → Tuple[shapely.strtree.STRtree, Dict[Tuple[tuple, ...], int]]

Helper function that builds an RTree from MultiPolygons

The Rtree is built from MultiPolygons using `shapely.strtree.STRtree`. Since the RTree returns the overlapping MultiPolygons, we need a way to retrieve the polygon's index. We do this with a dictionary from the exterior coordinates (*Polygon.exterior.coords*) of every Polygon in the MultiPolygon to the MultiPolygon's index in the provided Sequence.

Specifically, you can generate the key for a given MultiPolygon `mpoly` like so:

```
key = tuple([tuple(p.exterior.coords) for p in mpoly])
```

Parameters **polygons** – A Sequence of MultiPolygons. Must be iterable and able to be passed to the `STRtree` constructor. Iteration must be deterministic.

Returns A tuple of the RTree and the index mapping dictionary.

mobility_pipeline.lib.make_matrix.**make_a_to_b_matrix**(*a_cells: List[shapely.geometry.multipolygon.MultiPolygon], b_cells: List[shapely.geometry.multipolygon.MultiPolygon]*) → numpy.ndarray

Create an overlap matrix from sequence A to B

Computes for every pair of MultiPolygons between A and B, the fraction of the MultiPolygon in B that is covered by the one in A. We use an RTree to reduce the number of overlaps we have to compute by only computing overlaps between MultiPolygons that have overlapping bounding boxes.

Parameters

- **a_cells** – Sequence A of MultiPolygons
- **b_cells** – Sequence B of MultiPolygons

Returns A matrix with row indices that correspond to the indices of B and column indices that correspond to the indices of A. Every element at row `i` and column `j` in the matrix represents the fraction of the MultiPolygon in B at index `i` that overlaps with the MultiPolygon in A at index `j`.

mobility_pipeline.lib.make_matrix.**make_admin_admin_matrix**(*tower_tower: numpy.ndarray, tower_admin: numpy.ndarray, admin_tower: numpy.ndarray*) → numpy.ndarray

Compute the admin-to-admin matrix

Computed by multiplying the three provided matrices like so: (tower_admin) * (tower_tower) * (admin_tower)

Parameters

- **tower_tower** – The tower-to-tower mobility data
- **tower_admin** – Stores the fraction of each admin that is covered by each cell tower
- **admin_tower** – Stores the fraction of each cell tower's range that is within each admin

Returns An admin-to-admin mobility matrix such that each value with row index `i` and column index `j` is the estimated number of people who moved that day from the admin with index `i` to the admin with index `j`.

mobility_pipeline.lib.make_matrix.**make_admin_to_tower_matrix**(*admin_cells: List[shapely.geometry.multipolygon.MultiPo tower_cells: List[shapely.geometry.multipolygon.MultiPo* → numpy.ndarray

Compute the admin-to-tower matrix.

This is a wrapper function for *make_a_to_b_matrix()*, with matrices A and B as denoted for each argument.

Parameters

- **tower_cells** – Sequence of Voronoi cells; used as matrix A.
- **admin_cells** – Sequence of administrative regions; used as matrix B.

Returns The admin-tower matrix.

mobility_pipeline.lib.make_matrix.**make_tower_to_admin_matrix**(*tower_cells: List[shapely.geometry.multipolygon.MultiPo admin_cells: List[shapely.geometry.multipolygon.MultiPo* → numpy.ndarray

Compute the tower-to-admin matrix.

This is a wrapper function for *make_a_to_b_matrix()*, with matrices A and B as denoted for each argument.

Parameters

- **admin_cells** – Sequence of administrative regions; used as matrix A.
- **tower_cells** – Sequence of Voronoi cells; used as matrix B.

Returns The tower-admin matrix.

mobility_pipeline.lib.make_matrix.**make_tower_tower_matrix**(*mobility: pandas.core.frame.DataFrame, n_towers: int*) → numpy.ndarray

Make tower-to-tower mobility matrix

Thank you to Tomas Bencomo (https://github.com/tjbencomo) for writing the initial version of this function.

Parameters

- **mobility** – DataFrame of mobility data with columns `[ORIGIN, DESTINATION, COUNT]`. All values should be numeric.
- **n_towers** – Number of towers, which defines the length of each matrix dimension

Returns The tower-to-tower matrix, which has shape `(n_towers, n_towers)` and where the value at row `i` and column `j` is the mobility count for origin `i` and destination `j`.

mobility_pipeline.lib.overlap module

Utilities for working with overlapping Polygons and MultiPolygons

mobility_pipeline.lib.overlap.**compute_overlap**(*polygon_1: Union[shapely.geometry.polygon.Polygon, shapely.geometry.multipolygon.MultiPolygon], polygon_2: Union[shapely.geometry.polygon.Polygon, shapely.geometry.multipolygon.MultiPolygon]*)

Computes the fraction of the first polygon that intersects the second

The returned fraction is `(area of intersection) / (area of polygon_1)`.

Parameters

- **polygon_1** – The first polygon, whose total area will be the denominator for the computed fraction
- **polygon_2** – The second polygon

Returns The fraction of the first polygon that intersects the second

mobility_pipeline.lib.validate module

Functions for validating data file formats and contents

mobility_pipeline.lib.validate.**AREA_THRESHOLD = 0.0001**

Allowable deviance, as a fraction of the area of the union, between the area of the union of polygons and the sum of the polygons' individual areas. Agreement between these values indicates the polygons are disjoint and contiguous. Threshold was chosen based on the deviances in known good Voronoi tessellations.

mobility_pipeline.lib.validate.**all_numeric**(*string: str*) → bool

Check that a string is composed entirely of digits

Parameters **string** – String to check

Returns True if and only if the string is composed entirely of digits

mobility_pipeline.lib.validate.**validate_admins**(*country_id*) → Optional[str]

Check that the admins defined in the shapefile are reasonable

Admins are loaded using `load_admin_cells()`.

Checks:

- That the cells can be loaded by `load_admin_cells`.
- That the cells are contiguous and disjoint. This is checked by comparing the sum of areas of each polygon and the area of their union. These two should be equal.
- That at least one cell is loaded.

Parameters **country_id** – Country identifier.

Returns A description of a found error, or `None` if no error found.

mobility_pipeline.lib.validate.**validate_contiguous_disjoint_cells**(*cells: List[Union[shapely.geometry.multip shapely.geometry.polygon.Polygon]]*

Check that cells are contiguous and disjoint and that they exist

Checks:

- That the cells are contiguous and disjoint. This is checked by comparing the sum of areas of each polygon and the area of their union. These two should be equal. The allowable deviation is specified by *AREA_THRESHOLD*
- That at least one cell is loaded.

Returns A description of a found error, or `None` if no error found.

`mobility_pipeline.lib.validate.`**`validate_mobility`**(*raw: List[List[str]]*) → Optional[str]

Checks that the text from a CSV file is in a valid format for mobility

The text must consist of a list of rows, where each row is a list of exactly 4 strings: a date (not checked), an origin tower, a destination tower, and a count.

The origin and destination must be composed of digits following `data_interface.TOWER_PREFIX`. The count must be composed entirely of digits and represent a non-negative integer.

The origin and destination tower numeric portions must strictly increase in origin-major order.

Parameters **`raw`** – List of mobility CSV data by applying `list(csv.reader(f))`

Returns None if the input is valid, a string describing the error otherwise.

`mobility_pipeline.lib.validate.`**`validate_mobility_full`**(*mobility: List[List[str]]*) → Optional[str]

Check whether the mobility data file is correctly ordered and full

The mobility data file is loaded from the file at path *mobility_pipeline.data_interface.MOBILITY_PATH()*. Correctly ordered means that the tower names' numeric portions strictly increase in origin-major order. Full means that there is a row for every combination of origin and destination tower.

If this order were perfect, it would make forming the mobility matrix as easy as reshaping the last column. Unfortunately, this function showed that some coordinates are missing or out of order, so counts must be inserted manually.

Parameters **`mobility`** – List of mobility CSV data by applying `list(csv.reader(f))`

Returns None if there is no error, otherwise a description of the error.

`mobility_pipeline.lib.validate.`**`validate_tower_cells_aligned`**(*cells: List[shapely.geometry.multipolygon.MultiPoly*, *towers: numpy.ndarray*) → Optional[str]

Check that each tower's index matches the cell at the same index

For any cell `c` at index `i`, an error is found if `c` has nonzero area and the tower at index `i` is not within `c`.

Parameters

- **`cells`** – List of the cells (multi) polygons, in order
- **`towers`** – List of the towers' coordinates (latitude, longitude), in order

Returns A description of a found error, or `None` if no error found.

`mobility_pipeline.lib.validate.`**`validate_tower_index_name_aligned`**(*csv_reader: Iterator*) → Optional[str]

Check that in the towers data file, the tower names match their indices

Indices are zero-indexed from the second row in the file (to skip the header). An error is considered found if any tower name is not exactly `TOWER_PREFIX` appended with the tower's index.

Parameters **csv_reader** – CSV reader from calling `csv.reader(f)` on the open data file `f`

Returns A description of a found error, or `None` if no error found.

`mobility_pipeline.lib.validate.`**validate_voronoi**(*voronoi_path*) → Optional[str]

Check that the Voronoi cells are reasonable

Checks:

- That the cells can be loaded by `load_cells`.
- That the cells are contiguous and disjoint. This is checked by comparing the sum of areas of each polygon and the area of their union. These two should be equal.
- That at least one cell is loaded.

Returns A description of a found error, or `None` if no error found.

mobility_pipeline.lib.voronoi module

Tools for working with Voronoi tessellations

Given a 2-dimensional space with a set of points (called seeds), the Voronoi tessellation is a partitioning of the space such that for every partition, which is called a cell, the cell contains exactly one seed, and every point in the cell is closer to the cell's seed than it is to any other seed. For more information, see https://en.wikipedia.org/wiki/Voronoi_diagram.

class `mobility_pipeline.lib.voronoi.`**VoronoiCell**

Bases: `dict`

This class describes is for type hinting Voronoi cell JSONs

`mobility_pipeline.lib.voronoi.`**json_to_polygon**(*points_json: List[List[float]]*) → shapely.geometry.polygon.Polygon

Loads a Polygon from a JSON of points

Loads Polygon from a JSON of the format:

where each latitude-longitude pair describes a point defining the boundary of the polygon.

Parameters **points_json** – The points that define the boundary of the polygon

Returns A polygon

`mobility_pipeline.lib.voronoi.`**load_cell**(*cell_json: mobility_pipeline.lib.voronoi.VoronoiCell*) → shapely.geometry.multipolygon.MultiPolygon

Loads a Voronoi cell from JSON in `Polygon` or `MultiPolygon` format

Loads Voronoi cell from a JSON of the `Polygon` format:

or of the `MultiPolygon` format:

where each latitude-longitude pair describes a point of the Voronoi tessellation. If the JSON is in the `Polygon` format, a `shapely.geometry.MultiPolygon` object will be returned where the `MultiPolygon` has one member, the described polygon.

The value of the `type` key is used to distinguish `Polygon` and `MultiPolygon` formats.

Parameters **cell_json** – The points that define the boundary of the Voronoi cell

Returns A polygon of the Voronoi cell

Module contents

5.1.2 Submodules

5.1.3 mobility_pipeline.check_validation module

Script that checks the validity of data files

mobility_pipeline.check_validation.**validate_data_files**() → bool
Check the validity of data files

Note that these checks are computationally intensive, so they probably should not be included in an automated pipeline. Rather, they are for manual use.

Data files validated:

- Towers file at *mobility_pipeline.data_interface.TOWERS_PATH*
- Voronoi file at *mobility_pipeline.data_interface.VORONOI_PATH*
- Mobility file at *mobility_pipeline.data_interface.MOBILITY_PATH*

Returns `True` if all files are valid, `False` otherwise.

5.1.4 mobility_pipeline.data_interface module

Stores the constants and functions to interface with data files

This file is specific to the data files we are using and their format.

mobility_pipeline.data_interface.**ADMIN_ADMIN_TEMPLATE = 'data/brazil-towers-voronoi-mobilit**
Path to admin-to-admin matrix, accepts substitutions of country_id, day_id

mobility_pipeline.data_interface.**ADMIN_GEOJSON_TEMPLATE = 'data/brazil-towers-voronoi-mobi**
Path to admin GeoJSON file, accepts substitution of country_id

mobility_pipeline.data_interface.**ADMIN_SHAPE_PATH = 'data/brazil-towers-voronoi-mobility/ga**
Relative to py:const:*DATA_PATH*, path to administrative region shape file

mobility_pipeline.data_interface.**ADMIN_TOWER_TEMPLATE = 'data/brazil-towers-voronoi-mobilit**
Template that uses country identifier to make path to admin_tower matrix

mobility_pipeline.data_interface.**COUNTRY_ID = 'br'**
Country identifier

mobility_pipeline.data_interface.**DATA_PATH = 'data/brazil-towers-voronoi-mobility/'**
Path to folder containing towers, voronoi, and mobility data

mobility_pipeline.data_interface.**MOBILITY_PATH = 'data/brazil-towers-voronoi-mobility/mobi**
Relative to *DATA_PATH*, path to mobility CSV file

mobility_pipeline.data_interface.**TOWERS_PATH = 'data/brazil-towers-voronoi-mobility/towers_**
Relative to *DATA_PATH*, path to towers CSV file

mobility_pipeline.data_interface.**TOWER_ADMIN_TEMPLATE = 'data/brazil-towers-voronoi-mobilit**
Template that uses country identifier to make path to tower_admin matrix

mobility_pipeline.data_interface.**TOWER_PREFIX = 'br'**
The tower name is the tower index appended to this string

mobility_pipeline.data_interface.**VORONOI_PATH = 'data/brazil-towers-voronoi-mobility/brazil**
Relative to *DATA_PATH*, path to Voronoi JSON file

mobility_pipeline.data_interface.**convert_shape_to_json**(*shapefile_path_prefix: str*, *country_id: str*) → None
Converts shapefile containing administrative regions to GeoJSON format

The GeoJSON file is saved at ADMIN_GEOJSON_TEMPLATE % country_id

Parameters

- **shapefile_path_prefix** – Path to the .shp or .dbf shapefile, optionally without the file extension. Both the .shp and .dbf files must be present in the same directory and with the same name (except file extension).
- **country_id** – Unique identifier for the country and admin level.

Returns None

mobility_pipeline.data_interface.**deserialize_mat**(*mat_path: str*) → numpy.ndarray
Deserialize a matrix from a file

File must have been created by *serialize_mat()*.

Parameters **mat_path** – Path of matrix file

Returns Deserialized matrix

mobility_pipeline.data_interface.**load_admin_cells**(*identifier: str*) → List[shapely.geometry.multipolygon.MultiPolygon]
Loads the administrative region cells

Data is loaded from *ADMIN_GEOJSON_TEMPLATE* `% identifier`. This is a wrapper function for *load_polygons_from_json()*.

Returns A list of the administrative region cells.

mobility_pipeline.data_interface.**load_admin_tower**(*country_id: str*) → numpy.ndarray
Load admin-to-tower matrix

Data loaded from *ADMIN_TOWER_TEMPLATE* `% country_id`.

Parameters **country_id** – Country identifier

Returns The admin-to-tower matrix

mobility_pipeline.data_interface.**load_mobility**(*mobility_path: str*) → pandas.core.frame.DataFrame
Loads mobility data from the file at `mobility_path`.

Returns A `pandas.DataFrame` with columns `ORIGIN`, `DESTINATION`, and `COUNT`. Columns `ORIGIN` and `DESTINATION` contain numeric portions of tower names, represented as `numpy.int`. These numeric portions strictly increase in `ORIGIN`-major order, but rows may be missing if they would have had a `COUNT` value of `0`.

mobility_pipeline.data_interface.**load_polygons_from_json**(*filepath*) → List[shapely.geometry.multipolygon.MultiPolygon]
Loads cells from given filepath to JSON.

Returns A list of `shapely.geometry.MultiPolygon` objects, each of which describes a cell. If the cell can be described as a single polygon, the returned MultiPolygon will contain only 1 polygon.

mobility_pipeline.data_interface.**load_tower_admin**(*country_id: str*) → numpy.ndarray
Load tower-to-admin matrix

Data loaded from *TOWER_ADMIN_TEMPLATE* `% country_id`.

Parameters **country_id** – Country identifier

Returns The tower-to-admin matrix

mobility_pipeline.data_interface.**load_towers**(*towers_path: str*) → numpy.ndarray
Loads the tower positions from a file

Parameters **towers_path** – Path to towers file

Returns A matrix of tower coordinates with columns `[longitude, latitude]` and one tower per row. Row indices match the numeric portions of tower names.

mobility_pipeline.data_interface.**load_voronoi_cells**(*voronoi_path: str*) → List[shapely.geometry.multipolygon.MultiPolygon]
Loads cells

Parameters **voronoi_path** – Path to file to load cells from

Returns See `load_polygons_from_json`. Each returned object represents a Voronoi cell.

mobility_pipeline.data_interface.**save_admin_admin**(*country_id: str*, *day_id: str*, *admin_admin: numpy.ndarray*) → str
Save admin-to-admin matrix

Saved to *ADMIN_ADMIN_TEMPLATE* `% (country_id, day_id)`.

Parameters

- **country_id** – Country identifier
- **day_id** – Day identifier
- **admin_admin** – Admin-to-admin matrix to save

Returns Path at which matrix was saved

mobility_pipeline.data_interface.**save_admin_tower**(*country_id: str*, *mat: numpy.ndarray*) → None
Save admin-to-tower matrix

Saved to *ADMIN_TOWER_TEMPLATE* `% country_id`.

Parameters

- **country_id** – Country identifier
- **mat** – Matrix to save

Returns None

mobility_pipeline.data_interface.**save_tower_admin**(*country_id: str*, *mat: numpy.ndarray*) → None
Save tower-to-admin matrix

Saved to *TOWER_ADMIN_TEMPLATE* `% country_id`.

Parameters

- **country_id** – Country identifier
- **mat** – Matrix to save

Returns None

mobility_pipeline.data_interface.**serialize_mat**(*mat: numpy.ndarray*, *mat_path: str*) → None

Save a matrix to a file

Matrix is saved such that it can be recovered by *deserialize_mat()*.

Parameters

- **mat** – Matrix to save
- **mat_path** – File to save matrix to

Returns None

5.1.5 mobility_pipeline.gen_country_matrices module

Generate admin-to-tower and tower-to-admin matries for a country

mobility_pipeline.gen_country_matrices.**main**()

Main function called when script run

5.1.6 mobility_pipeline.gen_day_mobility module

Generate admin-to-admin mobility matrix for a single day

mobility_pipeline.gen_day_mobility.**main**()

Called when script run

5.1.7 mobility_pipeline.plot_voronoi module

Tool for plotting the Voronoi tessellation described by the provided data

Note that the seeds of the tessellation are based on the provided towers file, not computed from the cells. The tool also prints to the console the number of towers and number of cells. Towers without an associated cell are shown in green, while other towers are shown in red.

mobility_pipeline.plot_voronoi.**plot_polygon**(*axes: matplotlib.pyplot.axes*, *polygon: shapely.geometry.multipolygon.MultiPolygon*) → None

Add a polygon to an axes

Parameters

- **axes** – The axes to add the polygon to
- **polygon** – The polygon to add

Returns None

5.1.8 mobility_pipeline.visualize_overlaps module

Plots one Voronoi cell on top of all the administrative regions. The admins that intersect the Voronoi cell are colored by index and have the associated values from the tower-to-admin matrix printed. This lets you check that the matrix values seem reasonable.

mobility_pipeline.visualize_overlaps.**I_TOWER_TO_COLOR = 1**
Index of Voronoi cell to show.

mobility_pipeline.visualize_overlaps.**main**()
Main function that generates the plot

mobility_pipeline.visualize_overlaps.**plot_polygon**(*axes: matplotlib.pyplot.axes*, *polygon: shapely.geometry.multipolygon.MultiPolygon*, *color*, *_label=''*) → None
Plot a polygon (or multipolygon) with matplotlib

Parameters

- **axes** – The matplotlib axes to plot on
- **polygon** – The polygon to plot
- **color** – Color to use for shading the polygon
- **_label** – Label for the polygon that will be displayed in the legend

Returns None

5.1.9 Module contents

To get started, see our getting started guide. If you would like to contribute, see our contributing guide.

This project is hosted on GitHub at https://github.com/codethechange/mobility_pipeline

CHAPTER

SIX

PROJECT OVERVIEW

mobility_pipeline uses the relative geographical positions of cell towers and administrative regions (e.g. provinces) to transform mobility data describing how people move between cell towers into data on how people move between administrative regions. This lets us turn cell tower movement data that telecommunications providers already have into data on migration patters between political regions, which is what governments and NGOs need to plan disaster relief efforts.

CHAPTER

SEVEN

LEGAL

This project was created by Stanford Code the Change for UNICEF. It is available under the license in LICENSE.txt

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX

P

S

T

V