openelec Documentation

Release 0.0.4

Chris Arderne

Mar 28, 2019

Contents:

Documentation

openelec is a general tool for finding opportunities in electricity access. Able to create national-level plans for achieving universal electricity access, as well as optimise town/village-level mini-grid, densification and standalone systems. In addition, the tool provides functionality to find private-sector off-grid opportunities.

The library has a currently not very user-friendly Python API for scripting/notebook use. There is also a demonstration web interface running here (static front-end on AWS S3 with serverless backend on Lambda) and a short blog post here.

CHAPTER 1

National-level

A tool for modelling the optimal pathways to improving electricity access. Described in my blog post here: Modelling the optimum way to achieve universal electrification

CHAPTER 2

Town-level

A tool for optimising rural mini-grid systems and LV networks using OpenStreetMap building data and a minimum spanning tree approach to network optimisation. Described in my blog post here: A Flask app for mini-grid planning with a cost-optimised spanning tree

Web App usage (click to get proper resolution)

2.1 Model usage

To get to grips with the API and steps in the model, open the Jupyter notebook `example.ipynb`. This repository includes the input data needed to do a test run for Lesotho, so it should be a matter of opening the notebook and running all cells.

It also includes test data for a small village in central Lesotho to run the local version of the model.

2.2 Installation

Requirements

openelec requires Python >= 3.5 with the following packages installed:

- `flask` >= 1.0.2 (only for the web app)
- `numpy` >= 1.14.2
- `pandas` >= 0.22.0
- `geopandas` >= 0.4.0 (0.4.0 had API breaking changes so this version is needed)
- `shapely` >= 1.6.4
- `scipy` >= 1.0.0

- `scikit-learn` >= 0.17.1

Additionally these packages are needed for running the Jupyter notebook:

- `matplotlib`
- `jupyter`
- `folium`
- `descartes`

Install with pip

```
pip install openelec
```

Install from GitHub

Download or clone the repository and install the required packages (preferably in a virtual environment):

```
git clone https://github.com/carderne/openelec.git
cd gridfinder
pip install -r requirements.txt
```

You can run `./test.sh` in the directory, which will do an entire run through using the test data and confirm whether everything is set up properly.

2.3 Documentation

Built using sphinx. Available here. Build API reference with:

```
cd openelec/docscd
sphinx-apidoc -o . ../openelec
make html
```

2.3.1 API Reference

Submodules

openelec.clustering module

clusters module for openelec

Provides functions to read in a raster population dataset and convert to discrete vector polgons, each with a set population value. Additionally calculate each polygon's distance from a provided grid infrastructure vector.

Functions:

- prepare_clusters
- clip_rasters
- create_clusters
- filter_merge_clusters
- add_raster_layer
- add_vector_layer

- fix_column
- save_clusters

openelec.clustering.**add_raster_layer**(*clusters*, *raster*, *operation*, *col_name*, *affine=None*, *crs=None*)

The filter_merge_clusters() process loses the underlying raster values. So we need to use rasterstats.zonal_stats() to get it back.

Parameters

- **clusters** (*geopandas.GeoDataFrame*) – The processed clusters.
- **raster** (*str, pathlib.Path or numpy.ndarray*) – Either a path to the raster, or numpy.ndarray with the data.
- **operation** (*str*) – The operation to perform when extracting the raster data. Either 'sum', 'max', or 'mean'
- **col_name** (*str*) – Name of the column to add.
- **affine** (*affine.Affine(), optional*) – If a numpy ndarray is passed above, the affine is also needed.
- **crs** (*proj.crs, optional*) – Override raster's reported crs

Returns **clusters** – The processed clusters with new column.

Return type geopandas.GeoDataFrame

openelec.clustering.**add_vector_layer**(*clusters*, *vector*, *operation*, *col_name*, *shape*, *affine*, *raster_crs*)

Use a vector containing grid infrastructure to determine each cluster's distance from the grid.

Parameters

- **clusters** (*geopandas.GeoDataFrame*) – The processed clusters.
- **vector** (*str, pathlib.Path or geopandas.GeoDataFrame*) – Path to or already imported grid dataframe.
- **operation** (*str*) – Operation to perform in extracting vector data. Currently only 'distance' supported.
- **shape** (*tuple*) – Tuple of two integers representing the shape of the data for rasterizing grid. Sould match the clipped raster.
- **affine** (*affine.Affine()*) – As above, should match the clipped raster.

Returns **clusters** – The processed clusters with new column.

Return type geopandas.GeoDataFrame

openelec.clustering.**clip_raster**(*raster*, *boundary*, *boundary_layer=None*)

Clip the raster to the given administrative boundary.

Parameters

- **raster** (*string, pathlib.Path or rasterio.io.DataSetReader*) – Location of or already opened raster.
- **boundary** (*string, pathlib.Path or geopandas.GeoDataFrame*) – The poylgon by which to clip the raster.
- **boundary_layer** (*string, optional*) – For multi-layer files (like GeoPackage), specify the layer to be used.

Returns

Three elements:

clipped: numpy.ndarray Contents of clipped raster.

affine: affine.Affine() Information for mapping pixel coordinates to a coordinate system.

crs: dict Dict of the form {'init': 'epsg:4326'} defining the coordinate reference system of the raster.

Return type tuple

openelec.clustering.**create_clusters**(*raster*, *affine*, *crs*)

Create a polygon GeoDataFrame from the given raster

Parameters

- **raster** (*numpy.ndarray*) – The raster data to use.
- **affine** (*affine.Affine()*) – Raster pixel mapping information.
- **crs** (*dict*) – Dict of the form {'init': 'epsg:4326'} defining the coordinate reference system to use.

Returns **clusters** – A GeoDataFrame with integer index and two columns: geometry contains the Shapely polygon representations raster_val contains the values from the raster

Return type geopandas.GeoDataFrame

openelec.clustering.**filter_merge_clusters**(*clusters*, *max_block_size_multi=5*, *min_block_pop=50*, *buffer_amount=150*)

The vectors created by create_clusters() are a single square for each raster pixel. This function does the follows: - Remove overly large clusters, caused by defects in the input raster. - Remove clusters with population below a certain threshold. - Buffer the remaining clusters and merge those that overlap.

Parameters

- **clusters** (*geopandas.GeoDataFrame*) – The unprocessed clusters created by create_clusters()
- **max_block_size_multi** (*int, optional (default 5.)*) – Remove clusters that are more than this many times average size.
- **min_block_pop** (*int, optional (default 50.)*) – Remove clusters with below this population.
- **buffer_amount** (*int, optional (default 150.)*) – Distance in metres by which to buffer the clusters before merging.

Returns **clusters** – The processed clusters.

Return type geopandas.GeoDataFrame

openelec.clustering.**fix_column**(*clusters*, *col_name*, *factor=1*, *minimum=0*, *maximum=None*, *no_value=None*, *per_capita=False*)

A number of operations to apply to a columns values to get desired output.

Parameters

- **clusters** (*GeoDataFrame*) – The clusters object.
- **col_name** (*str*) – The column to apply the operation to.
- **factor** (*float, optional (default 1.)*) – Factor by which to multiply the column vales.

- **minimum** (*float, optional (default 0.)*) – Apply a minimum threshold to the values.
- **maximum** (*str, optional*) – Currently only supported for 'largest'. Limits the values to double the value of the cluster with the highest population.
- **no_value** (*str, optional*) – Currently only supported for 'median'. Replaces NaN instances with the median value.
- **per_capita** (*boolean, optional (default False.)*) – Divide values by cluster population.

Returns **clusters** – The 'fixed' clusters.

Return type GeoDataFrame

openelec.clustering.**prepare_clusters**(*country*, *ghs_in*, *gdp_in*, *travel_in*, *ntl_in*, *aoi_in*, *grid_in*, *clusters_out*)

Run all.

openelec.clustering.**save_clusters**(*clusters*, *out_path*)

Convert to EPSG:4326 and save to the specified file. clusters: geopandas.GeoDataFrame

The processed clusters.

out_path: str or pathlib.Path Where to save the clusters file.

openelec.conv module

Module for loading and saving.

Functions:

- read_data
- merge_geometry
- spatialise
- geojsonify
- geometry
- properties
- overpass
- json2geojson

-save_to_path

openelec.conv.**geojsonify**(*gdf*, *property_cols=[]*)

Convert GeoDataFrame to GeoJSON that can be supplied to JavaScript.

Parameters

- **gdf** (*geopandas.GeoDataFrame*) – GeoDataFrame to be converted.
- **property_cols** (*list, optional*) – List of column names from gdf to be included in 'properties' of each GeoJSON feature.

Returns **geoJson** – A GeoJSON representatial List of column names from gdf to be included in 'properties' of each GeoJSON feature.

Return type dict

Returns **geojson** – A GeoJSON representation that can be parsed by standard JSON readers.

Return type dict

openelec.conv.**geometry**(*coordinates*)
Convert a GeoDataFrame geometry value into a GeoJSON-friendly form..

Parameters **coords** (*shapely.LineString, shapely.Polygon or shapely.MultiPolygon*) – A single geometry entry from a GeoDataFrame.

Returns

geom_dict – A GeoJSON geometry element of the form 'geometry': {

'type': type, 'coordinates': coords

}

Return type dict

openelec.conv.**json2geojson**(*items*)
Convert a json from OSM Overpass to a GeoJSON.

Parameters **items** (*json*) – The JSON representation.

Returns **geojson** – As a GeoJSON.

Return type dict

openelec.conv.**merge_geometry**(*results*, *geometry*, *columns=None*)
Merge results from modelling with an original input geometry, using the index as key for both.

Parameters

- **results** (*list of dicts*) – Output from modelling.
- **geometry** (*GeoDataFrame*) – Target geometry with amtching index.
- **columns** (*list, optional*) – List of columns to include in output. If not provided, keep all.

Returns **spatial** – Results with geometry.

Return type GeoDataFrame

openelec.conv.**overpass**(*bounds*)
Get OSM Overpass results from specified bounds as GeoJSON.

Parameters **bounds** (*str*) – String in this format: "S, W, N, E"

Returns **geojson** – GeoJSON results.

Return type dict

openelec.conv.**properties**(*row*, *property_cols*)
Get the selected columns from the pandas row as a GeoJSON-friendly dict.

Parameters

- **row** (*pandas.Series*) – A single row from a GeoDataFrame.
- **property_cols** (*list*) – List of column names to be added.

Returns

prop_dict – A GeoJSON element of the form properties: {

'column1': property1, 'column2': property2, ...

}

Return type dict

openelec.conv.**read_data**(*data*)
Read targets (clusters, buildings) data from a file or other source.

Parameters **data** (*Path, str, dict*) – Path to a Fiona-readable file, or GeoJSON-like dict. Can also be a string representation of a GeoJSON.

Returns **targets** – The data filtered and processed.

Return type GeoDataFrame

openelec.conv.**save_to_path**(*path*, ***features*)
Save the provided features in the directory specified. File names are taken from the keywords.

openelec.conv.**spatialise**(*results*, *type='line'*)
Convert results to a GeoDataFrame using values to create a geometry.

Parameters

- **results** (*list of dicts*) – An output from the modelling.
- **type** (*str, optional (default 'line'.)*) – What type of geometry it is. (Currently only implemented for 'line').

Returns **spatial** – Results with geometry.

Return type GeoDataFrame

openelec.local module

local module for openelec Tool designed to take a small village and estimate the optimum connections, based on a PV installation location and economic data.

Includes LocalModel class and calculate_profit function.

class openelec.local.**LocalModel**(*data*)
Bases: *openelec.model.Model*

Inherits from Model. Goal is to fully merge NationalModel and LocalModel, as they share lots of functionality.

This class provides most of the functionality for using openelec at the local level.

baseline(*min_area=20*)
Filter on population and assign whether currently electrified.

Parameters

- **targets** (*GeoDataFrame*) – Loaded targets.
- **min_area** (*int, optional (default 20.)*) – Minimum target area in m2.

connect_targets(*origin=None*)
Create an MST connecting the target features.

Parameters **origin** (*tuple of two floats*) – Tuple of format (latitude, longitude) of origin point (such as a generator) if desired. If not supplied, the largest target building is used instead,

model(*target_coverage=None*)
Run the model with the given economic parameters and return the processed network and nodes.

cut arcs one by one, see which cut is the *most* profitable, and then take that network and repeat the process annual income should be specified by the nodes

Then we start with the complete network, and try 'deleting' each arc. Whichever deletion is the most profitable, we make it permanent and repeat the process with the new configuration. This continues until there are no more increases in profitability to be had.

Parameters **target_coverage** (*float, optional*) – If provided, model will aim to achieve this level of population coverage, rather than optimising on NPV.

parameters(*demand*, *tariff*, *gen_cost*, *cost_wire*, *cost_connection*, *opex_ratio*, *years*, *discount_rate*)
Set up model parameters.

Parameters

- **demand** (*int*) – Demand in kWh/person/month.
- **tariff** (*float*) – Tariff to be charged in USD/kWh.
- **gen_cost** (*int*) – Generator cost in USD/kW.
- **cost_wire** (*int*) – Wire cost in USD/m.
- **cost_connection** (*int*) – Cost per household connection in USD.
- **opex_ratio** (*float*) – Annual OPEX as a percentage of CAPEX (range 0 -1).
- **years** (*int*) – Project duration in years.
- **discount_rate** (*float*) – Discount rate to be used for NPV calculation (range 0-1).

spatialise()
Convert all model output to GeoDataFrames.

summary()
Calculate some quick summary numbers for the village.

Returns **results** – Dict of summary results.

Return type dict

openelec.local.**calculate_profit**(*network*, *nodes*, *index*, *disabled_arc_index*, *cost*, *income_per_month*, *cost_wire*, *cost_connection*, *num_people_per_m2*, *demand*, *tariff*)
Here we recurse through the network and calculate profit, starting with all arcs that connect to the index node, and get the end-nodes for those arcs calculate profit on those nodes, and then recurse! disabled_arc should be treated as if disabled

Parameters

- **nodes** (*network,*) – Current state of both.
- **index** (*int*) – Current index.
- **disabled_arc_index** (*int*) – Arc that is currently disabled.
- **etc** (*cost*) –

openelec.model module

model module for openelec.

Provides common functionality for LocalModel and NationalModel.

class openelec.model.**Model**(*data*)
Bases: object

Base class for NationalModel and LocalModel.

baseline()

results_as_geojson(*network_columns=None*, *targets_columns=None*)
Convert all model output to GeoJSON.

save_to_path(*path*)
Save the resultant network and buildings to GeoJSON files. spatialise() must have been run before.

Parameters **path** (*str, Path*) – Path to a directory to create GeoJSON files. Will be created if needed, will not prompt on overwrite.

setup(*sort_by=None*, ***kwargs*)
Basic set up on target features.

Parameters

- **min_area** (*int, optional (default 20.)*) – Area in m2, below which features will be excluded.
- ****kwargs** (**dict) – see baseline()

openelec.national module

national module for openelec

Includes NationalModel class and find_best function.

class openelec.national.**NationalModel**(*data*)
Bases: *openelec.model.Model*

Inherits from Model. Goal is to fully merge NationalModel and LocalModel, as they share lots of functionality.

This class provides most of the functionality for using openelec at the national level.

baseline()
Filter on population and assign whether currently electrified.

connect_targets()
Create an MST connecting the target features.

demand_levels()
Calculate demand level in kWh/p/month, either from MTF or using a simple formula.

TODO Add productive use, schools

dynamic(*steps=4*, *years_per_step=5*)
Run the model dynamically, splitting into a specified number of steps with a number of years between each one. Creates an iterator that yields results after each step.

Parameters

- **steps** (*int, optional (default 4.)*) – Number of steps to use.
- **years_per_step** (*int, optional (default 5.)*) – Number of years per step.
- **demand_factor** (*int, optional (default None.)*) – If provided, uses this factor in demand calculations. If None, uses the MTF levels instead.

Yields

- **targets_out, networks_out** (*GeoDataFrames*) – The next step of targets and network.
- **results** (*dict*) – The next step of results.

dynamic_combine()
Run the dyamic model and combine the results into a single set of GeoDataFrames and a results dict.

Returns

- **targets, network** (*GeoDataFrames*) – Combined targets and network.
- **results** (*dict*) – Dict of results keyed on step number.

initial_access()
Calibrate initial electricity access levels (per electrified cluster) to match national statistics.

model()
Run the national planning model with the provided parameters.

Then we're ready to calculate the optimum grid extension. This is done by expanding out from each already connected node, finding the optimum connection of nearby nodes. This is then compared to the off-grid cost and if better, these nodes are marked as connected. Then the loop continues until no new connections are found.

parameters(*grid_mv_cost=50*, *grid_lv_cost=3*, *grid_trans_cost=3500*, *grid_conn_cost=200*, *grid_opex_ratio=0.02*, *mg_gen_cost=4000*, *mg_lv_cost=2*, *mg_conn_cost=100*, *mg_opex_ratio=0.02*, *actual_pop=10000000.0*, *pop_growth=0.01*, *access_tot=0.3*, *access_urban=0.66*, *grid_dist_connected=2*, *minimum_pop=100*, *gdp_growth=0.02*, *discount_rate=0.08*, *people_per_hh=5*, *target_access=1.0*, *demand_factor=5*, *use_mtf=False*)
Set up model parameters

spatialise(*filter_network=True*)
Basic filtering and processing on results. Targets 'type' can be one of: - densify: was always connected - grid: new grid connection - offgrid: new off-grid connection TODO Add no-connection type.

Parameters **targets** (*network,*) – Output from model.

summary(*to_densify=None*)
Calculate some summary results.

Returns **results** – Dict of summary results.

Return type dict

openelec.national.**find_best**(*network*, *nodes*, *index*, *prev_arc*, *b_demand=0*, *b_length=1e-09*, *b_nodes=[]*, *b_arcs=[]*, *c_demand=0*, *c_length=1e-09*, *c_nodes=[]*, *c_arcs=[]*)
This function recurses the network, bringing current **c_** values with it. These aren't returned, so are left untouched side-branch explorations. The **b_** values are returned, and updated when a better configuration found. Thus these will remember the best solution including all side meanders.

openelec.network module

network module of openelec. Provides functionality for creating MST and network from input points.

Functions:

- create_network
- spanning_tree
- add_origin

- remove_existing
- direct_network

openelec.network.**add_origin**(*points*, *origin*)

If origin not specified, the model defaults to using index 0 as the 'main' point. Thus targets should already have been sorted by population/area with largest first.

openelec.network.**create_network**(*targets*, *columns*, *existing_network=False*, *directed=False*, *origin=None*)

We then take all the clusters and calculate the optimum network that connects them all together. We use this to create a graph network of and nodes and arcs representing the clusters and connections.

Parameters

- **clusters** (*GeoDataFrame*) – The prepared clusters.
- **columns** (*list*) – List of columns to include.
- **existing_network** (*boolean, optional (default False.)*) – Whether there is an existing network. In this case, redundant lines in new network are removed.
- **directed** (*boolean, optional (default False.)*) – Whether the output network should be directed.
- **origin** (*tuple, optional*) – Location of an origin (e.g. generator) for the network. Should be of the form (latitude, longitude) in WGS84 coordinates. If not provided, the first element is set as origin.

Returns

- **network** (*list of dicts*) – The network arcs.
- **nodes** (*list of dicts*) – The network nodes.

openelec.network.**direct_network**(*network*, *nodes*, *index*, *prev*)

Recursive function to direct the network from the PV point outwards We need to calculate the directionality of the network, starting from the PV location and reaching outwards to the furthest branches.

We use this to calculate, for each node, it's marginal and total distance from the PV location. At the same time, we tell each arc which node is 'upstream' of it, and which is 'downstream'. We also tell each node which arcs (at least one, up to three or four?) it is connected to.

Parameters

- **network** (*list of dicts*) – Containing the arc representations.
- **nodes** (*list of dicts*) – Containing the building node representations.
- **index** (*int*) – Current node index that we're looking at.

Returns

- **network** (*list of lists*) – Nearby network directed for current node.
- **nodes** (*list of list*) – The nodes object.

openelec.network.**remove_existing**(*network*, *nodes*)

Set which arcs don't already exist (and the remainder do!)

openelec.network.**spanning_tree**(*X*, *approximate=False*)

Function to calculate the Minimum spanning tree connecting the provided points X. Modified from astroML code in mst_clustering.py

Parameters **X** (*array_like*) – 2D array of shape (n_sample, 2) containing the x- and y-coordinates of the points.

Returns **x_coords, y_coords** – the x and y coordinates for plotting the graph. They are of size [2, n_links], and can be visualized using `plt.plot(x_coords, y_coords, '-k')`

Return type ndarrays

openelec.prioritise module

prioritising module for openelec

Functions:

- priority

openelec.prioritise.**priority**(*clusters*, *pop_range=None*, *grid_range=None*, *ntl_range=None*, *gdp_range=None*, *travel_range=None*)

Calculate the priority clusters that meet the criteria, and calculate a score from 1-5 for each.

Parameters

- **clusters** (*GeoDataFrame*) – Village clusters object.
- **min_grid_dist** (*int*) – Minimum distance from grid in metres to consider for clusters.
- **max_ntl** (*int*) – Maximum value of NTL (night time lights) to consider. Range 0-255.

Returns

- **clusters** (*GeoDatFrame*) – Processed clusters.
- **summary** (*dict*) – Summary results.

openelec.util module

Helper functions for models.

Functions:

- connect_houses
- stranded_arcs
- calc_coverage
- assign_coverage
- calc_lv

openelec.util.**assign_coverage**(*targets*, *access_rate*)

Estimate level of electricity access for each target.

openelec.util.**calc_coverage**(*weight*, *pop*, *conn*, *pop_tot*, *target_access*, *accuracy=0.01*, *increment=0.1*, *max_coverage=0.8*)

Estimate coverage levels for the given parameters.

Parameters

- **pop, conn** (*weight,*) – Each a column from targets.
- **pop_tot** (*int*) – Total population.
- **target_access** (*float*) – Target access rate.
- **accuracy** (*float, optional (default 0.01.)*) – Acceptable accuracy level.

- **increment** (*float, optional (default 0.1.)*) – How much to increment each target's coverage level on each loop.
- **max_coverage** (*float, optional (default 0.8.)*) – Max coverage level for any target.

Returns **coverage** – Array of coverage levels of same shape as weight.

Return type numpy array

openelec.util.**calc_lv**(*people*, *demand*, *people_per_hh*, *area*)
Calculate LV cost parameters for the given parameters. Everything is in m and m2.

openelec.util.**connect_houses**(*network*, *nodes*, *index*)
Then we disconnect all the houses that are no longer served by active arcs, and prune any stranded arcs that remained on un-connected paths. now we need to tell the houses that aren't connected, that they aren't connected (or vice-versa)

Start from base, follow connection (similar to calculate_profit) and swith node[6] to 1 wherever connected and only follow the paths of connected houses.

openelec.util.**stranded_arcs**(*network*, *nodes*)
And do the same for the stranded arcs

Module contents

openelec package contains the following modules:

- clustering
- model
- local
- national
- io
- network
- prioritise
- util

GPL-3.0 (c) Chris arderne

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

o

Index

P

R

S