Service Area Tools

1 Introduction

When planning services, whether they be state-ran or commercial that involve engagement with the public such as healthcare services or supermarkets, the location matters. For supermarkets this involves competitor analysis or identifying communities which lack suitable options (Suárez-Vega *et al*. 2012). For healthcare services, this involves ensuring and maintaining good accessibility across the population and has been the focus of numerous studies, which have been key in identifying underserved communities and links with deprivation links with deprivation (Comber, *et al*. 2011; Foley and Darby, 2002). In these examples, and others similar to them, geographic distance forms a major component of the studies.

Existing easy access and free to use network analysis tools such as those found within QGIS (QGIS, 2024) often create network graphs by scaling down the density of nodes and edges present. Whilst reducing the density of nodes and edges in the graph increases the speed of the path-finding process, this comes at the expense of high-resolution path-finding, which can have significant impacts in dense, complex urban road networks. In addition to this, popular, free and open-access network path-finding tools which retain the complexity of road networks do not currently iterate over multiple start and or end locations without modification.

Using open source packages, the tools developed within Service Area Tools focus on rapidly creating network graphs, whilst retaining the maximum resolution of the road network using OpenStreetMap (OSM) data (OpenStreetMap, 2024). The current scope of the project includes the automation of network graph creation, service areas (maximum distance travelable) and creation of service bands also known as isochrones through dissolving and differencing overlapping service areas.

Ongoing development involves the precise calculation of the shortest distance between each start and end location, however this has proven problematic to scale up to scalable due to it being an *extremely* computationally expensive task and is currently a WIP; use with caution.

The tools were built primarily around the use of the pyrosm, network, osmnx and geopandas amongst others. A full list is provided within the following sections and also available in the environment.yml file.

The repository is hosted on GitHub at: https://github.com/hularuns/Service-Area-Tools

2 Setup

2.1 Cloning the Repository

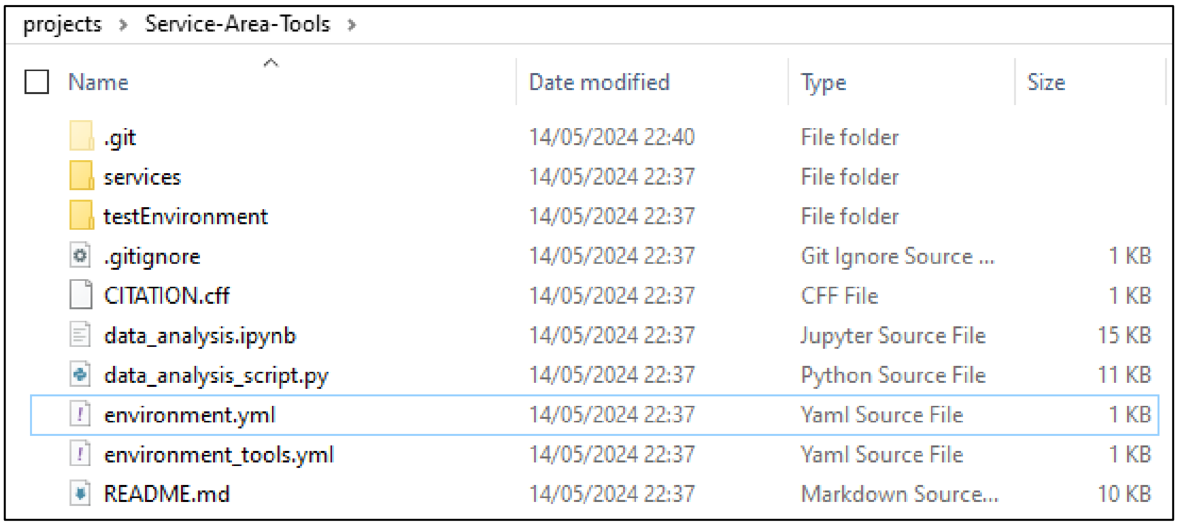
Using Git Bash, navigate to the directory you wish to clone the repository to, for example:

```
cd C:/Users/[username]/projects
```

Clone the repository, which will create a folder called **/Service-Area-Tools/** with the command:

```
git clone https://github.com/hularuns/Service-Area-Tools
```

If successful, the folder will have cloned and the contents will look similar to Figure 1 below (correct as of v1.0.3)



projects > Service-Area-Tools >

Name	Date modified	Type	Size
.git	14/05/2024 22:40	File folder	
services	14/05/2024 22:37	File folder	
testEnvironment	14/05/2024 22:37	File folder	
.gitignore	14/05/2024 22:37	Git Ignore Source ...	1 KB
CITATION.cff	14/05/2024 22:37	CFF File	1 KB
data_analysis.ipynb	14/05/2024 22:37	Jupyter Source File	15 KB
data_analysis_script.py	14/05/2024 22:37	Python Source File	11 KB
environment.yml	14/05/2024 22:37	Yaml Source File	1 KB
environment_tools.yml	14/05/2024 22:37	Yaml Source File	1 KB
README.md	14/05/2024 22:37	Markdown Source...	10 KB

Figure 1. Cloned repository contents as of v1.0.0.

2.2 Setting Up Your Python Environment

The following section will install all necessary dependencies for both the use of the tools and also the example script.

2.2.1 Using pip

Service Area Tools can be installed using pip to an existing virtual environment by navigating to the **/Service-Area-Tools/** folder using terminal, shell or bash run the following command:

```
pip install "python>=3.9" "geopandas<=0.14.3" "pandas<=2.2.2"
networkx ipykernel matplotlib pyrosm alphashape faker folium jupyter
tqdm
```

Should you only wish to install the bare minimums for running the basic tools exclusive of the example script, run the following command:

```
pip install "python>=3.9" "geopandas<=0.14.3" "pandas<=2.2.2"
networkx matplotlib pyrosm alphashape faker tqdm
```

2.2.2 Using Conda

Ensure that you have conda installed; it is available here **here** for download.

Navigate to the cloned repository folder using the conda terminal, create a new conda environment with the following command in the conda terminal:

This creates a conda environment called netgeo_env with all dependencies.

```
conda env create -f environment.yml
```

Or if you only want an environment which only includes the bare minimum for running the tools:

This creates a conda environment called netgeo_env_tools with all necessary dependencies for base functionality of tools within **services/**.

```
conda env create -f environment_tools.yml
```

2.3 Running Example Script

2.3.1 From Command Line – Python Script

The recommended way to run the example script is using the terminal. With the **netgeo_env** environment enabled, navigate to the root folder of the repository and run the following:

```
python data_analysis_script.py
```

2.3.2 Within an IDE

The recommended way to run the example script (data_analysis.ipynb) is using an IDE such as VSCode which supports the use of jupyter notebooks through additional plugins. Once the jupyter notebook plugin is installed, it is as simple as loading the notebook file within the IDE, selecting the **netgeo_env** python interpreter and pressing **Run All**.

2.3.3 From Command Line – Jupyter Notebook

Alternatively, the example script can be ran directly from the conda terminal or an IDE such as VScode with jupyter notebook plugins installed.

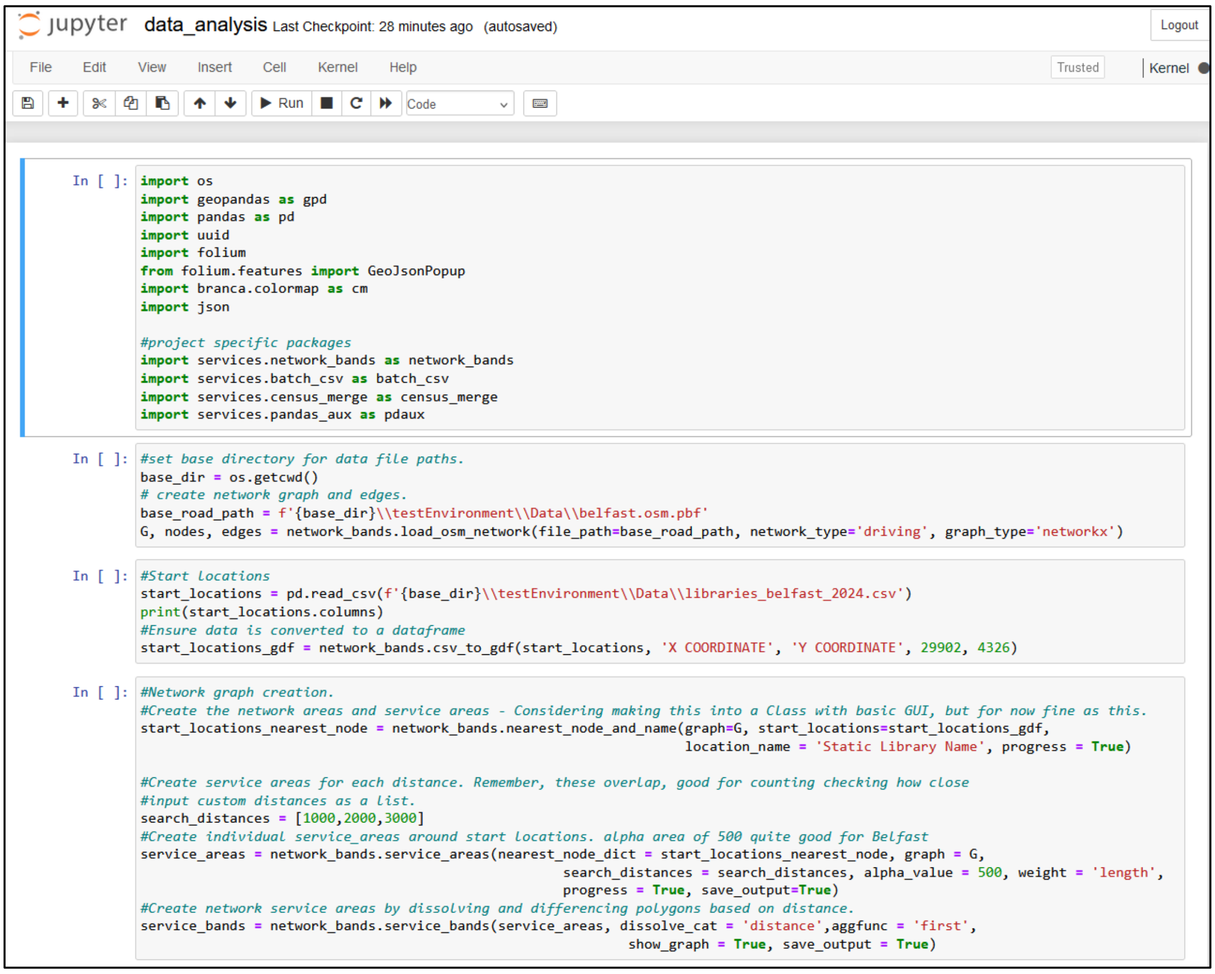
In the conda terminal, ensure that the **netgeo_env** environment is activated:

```
activate netgeo_env
```

Navigate to the cloned repository and run the notebook file:

```
jupyter notebook data_analysis.ipynb
```

If successful, a browser should open with the option which should look similar to Figure 2.


jupyter data_analysis Last Checkpoint: 28 minutes ago (autosaved)
Logout
File Edit View Insert Cell Kernel Help
Trusted
Kernel
Run
Code

```
In [ ]: import os
        import geopandas as gpd
        import pandas as pd
        import uuid
        import folium
        from folium.features import GeoJsonPopup
        import branca.colormap as cm
        import json

        #project specific packages
        import services.network_bands as network_bands
        import services.batch_csv as batch_csv
        import services.census_merge as census_merge
        import services.pandas_aux as pdaux

In [ ]: #set base directory for data file paths.
        base_dir = os.getcwd()
        # create network graph and edges.
        base_road_path = f'{base_dir}\\testEnvironment\\Data\\belfast.osm.pbf'
        G, nodes, edges = network_bands.load_osm_network(file_path=base_road_path, network_type='driving', graph_type='networkx')

In [ ]: #Start locations
        start_locations = pd.read_csv(f'{base_dir}\\testEnvironment\\Data\\libraries_belfast_2024.csv')
        print(start_locations.columns)
        #Ensure data is converted to a dataframe
        start_locations_gdf = network_bands.csv_to_gdf(start_locations, 'X COORDINATE', 'Y COORDINATE', 29902, 4326)

In [ ]: #Network graph creation.
        #Create the network areas and service areas - Considering making this into a Class with basic GUI, but for now fine as this.
        start_locations_nearest_node = network_bands.nearest_node_and_name(graph=G, start_locations=start_locations_gdf,
                                           location_name = 'Static Library Name', progress = True)

        #Create service areas for each distance. Remember, these overlap, good for counting checking how close
        #input custom distances as a list.
        search_distances = [1000,2000,3000]
        #Create individual service_areas around start locations. alpha area of 500 quite good for Belfast
        service_areas = network_bands.service_areas(nearest_node_dict = start_locations_nearest_node, graph = G,
                                                    search_distances = search_distances, alpha_value = 500, weight = 'length',
                                                    progress = True, save_output=True)
        #Create network service areas by dissolving and differencing polygons based on distance.
        service_bands = network_bands.service_bands(service_areas, dissolve_cat = 'distance',aggfunc = 'first',
                                                    show_graph = True, save_output = True)
```

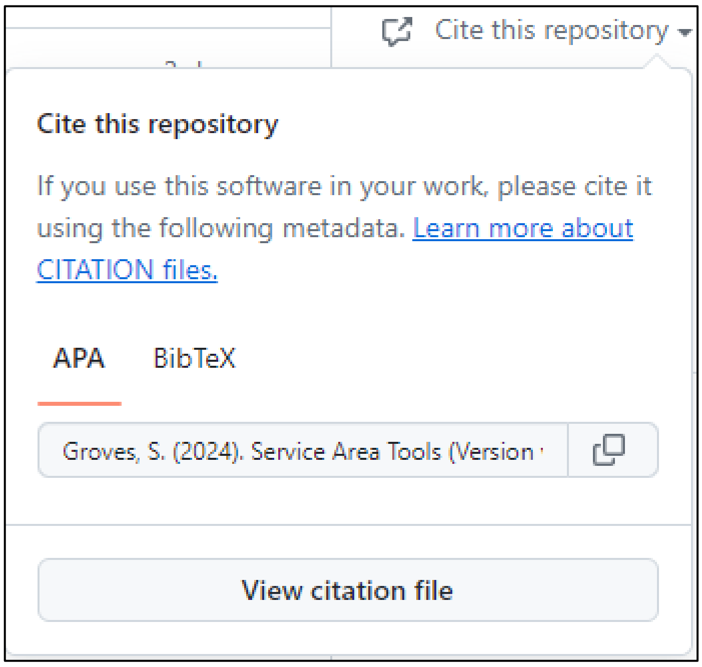
Figure 2. Example script ran from command line.

All outputs will create a service_areas.gpkg, service_bands.gpkg and test.html which are explained further within the **Usage** section.

2.4 Use In Your Own Project

Should you wish to use the tools within your own project, you simply have to include the **/services/** folder, and ensure that your project has all the dependencies installed (**environment_tools.yml** will be the most up-to-date source list).

If you do choose to use it in your own project, we kindly ask that you cite the repository in your own projects using the metadata in **CITATION.cff** or from the citation information on GitHub (Figure 3)



Cite this repository

If you use this software in your work, please cite it using the following metadata. Learn more about CITATION files.

Groves, S. (2024). Service Area Tools (Version

Figure 3. Example of repository citation.

3 Usage

This section will describe in detail the functionality of the tools within the repository. See the example script (**data_analysis.ipynb/data_analysis_script.py**) and test data within **testenvironment/Data** for a working example of the core functionality of the service area creations with a basic example of subsequent data analysis. **network_bands** forms the core functionality of Service-Area-Tools. **batch_csv**, **census_merge** and **pandas_aux** are ancillary scripts which are used in the creation of the example script to enable the data analysis after the creation of the service areas and service bands. The ancillary scripts contain functions will likely be useful for the types of data analysis typical of service areas.

All functions within this project have detailed docstrings which can be accessed via the **help()** function or **__doc__**. For example:

```
from services import network_bands

#using __doc__
network_bands.nearest_node_and_name.__doc__
# using help()
help(network_bands.nearest_node_and_name)
```

3.1.1 Simplified Example Workflow

The following code block is a bare bones simplified working example of how to use the core functionality to create service areas and service bands with the data supplied in **testenvironment/Data**.

```
from services import network_bands
import pandas as pd
import os

#1)create graph and extract nodes and edges.
base_dir = os.getcwd()
# create network graph and edges.
base_road_path = f'{base_dir}\\testEnvironment\\Data\\belfast.osm.pbf'

G, nodes, edges = network_bands.load_osm_network(file_path = file_path =
base_road_path, network_type = 'driving', graph_type='networkx')

#2)load start_locations csv
start_locations =
pd.read_csv(f'{base_dir}\\testEnvironment\\Data\\libraries_belfast_2024.csv'


#3) convert to csv to geodataframe
start_locations_gdf = network_bands.csv_to_gdf(csv = start_locations, x_col =
'X COORDINATE', y_col = 'Y COORDINATE',
input_crs = 29902, crs_conversion = 4326)

#4)calculate the start_locations nearest node on the graph
start_locations_nearest_node = network_bands.nearest_node_and_name(graph = G,
start_locations = start_locations_gdf,location_name = 'Static Library Name')

#5) calculate service_areas
search_distances = [1000,2000,3000] #define distances to search by.

#6) create service areas
service_areas = network_bands.service_areas(nearest_node_dict =
start_locations_nearest_node, graph = G, search_distances = search_distances,
alpha_value = 500, weight = 'length',
save_output = True)
'service area polygons have been successfully saved to a geopackage'

#7) create service bands by differencing and dissolving
service_bands = network_bands.service_bands(service_areas, dissolve_cat =
'distance',aggfunc = 'first', show_graph = True, save_output = True)
'Network areas have successfully been dissolved and differenced'
-------matplotlib_plot (figure 1)------
'A map showing network contours has been created. '
'service area polygons have been successfully saved to a geopackage'
```

3.2 network_bands

There are five functions found within **network_bands.py** which form the basic expected workflow and provide the core functionality of the tools which automate the creation of road network service areas.

3.2.1 load_osm_network()

```
load_osm_network(file_path:str, network_type:str, graph_type:str)
```

This function loads an offline OSM .pbf file which can be downloaded using **osmconvert** or **Osmosis**, creating a network using the networkx, pandana or igraph packages (in the example, networkx is used and tested and therefore recommended). The function returns a MultiDiGraph and its nodes and edges as geodataframes. The output CRS is usually EPSG: 4326, as is typical of OSM data.

Parameters:

- file_path (string): A path string to the .pbf OSM road network.
- network_Type (string): Type of network to generate, full argument list available in networkx, pandana or igraph documentation.
- graph_type (string): Type of graphs to create, available types = network, ipgraph or pandana.

Example:

```
from services import network_bands

roads = 'C:/Users/user/projects/Service-Area-Tools/belfast.osm.pbf'

G, nodes, edges = network_bands.load_osm_network(file_path = roads, >>>
network_type = 'driving', graph_type = 'networkx')
```

3.2.2 csv_to_gdf()

```
csv_to_gdf(csv, x_col:str, y_col:str, input_crs:int, crs_conversion:int =
None)
```

This function georeferences a CSV with separate X and Y coordinate columns and converts it to a geodataframe with a geometry in an assigned coordinate reference system (CRS). There is additionally an option to convert the CRS.

Example:

```
from services import network_bands

gdf = network_bands.csv_to_gdf(csv = csv, x_col = 'X COORDINATE', y_col = 'Y
COORDINATE', input_crs = 29902, crs_conversion = 4326)
```

Parameters:

- csv (string): CSV file with location names or unique IDs with separate X and Y columns.
- x_col (string): Name of column with X-coordinates.
- y_col (string): Name of column with Y-coordinates.
- input_crs (int): CRS of coordinates of x_col and y_col.
- crs_conversion (int): Optional – CRS to convert the geometries to.

3.2.3 nearest_node_and_name()

```
nearest_node_and_name(graph, locations: gpd.GeoDataFrame, location_name: str =
None, anon_name: bool = False)
```

This function calculates the nearest node on the network graph which the input location is closest to. The output creates a dictionary of dictionaries for each location, with the end result looking similar to:

```
    >>>  {'Ardoyne Library': {'nearest_node': 475085580},
    >>> 'Ballyhackamore Library': {'nearest_node': 73250694},
    >>> 'Belfast Central Library': {'nearest_node': 4513699587}}
```

If no `location_name` = `None`, then the function will create names based off of index, e.g. location_1, location_2 etc.

The function works with the outputs of **load_osm_network()** and **csv_to_gdf()** functions seamlessly.

Parameters:

- graph (MultiDiGraph): Network graph, for example one created with network.
- locations (GeoDataFrame): Geodataframe of start locations with unique ID or name column.
- location_name (str): Column with unique IDs or names.
- anon_name (bool): If True, fake names will be generated using Faker.

Example:

```
from services import network_bands

node_dict = nearest_node_and_name(graph = G, start_locations = gdf,
location_name = name, anon_name = False)
```

3.2.4 service_areas()

```
service_areas(nearest_node_dict:dict, graph, search_distances:list,
alpha_value:int, weight:str, save_output:bool = False)
```

This function creates service areas around the start points from a dictionary of nearest nodes, such as that generated by **nearest_node_and_name()**. Each distance specified is iterated over for each location. A list of reachable nodes within each of these specified distances is created and alpha shapes are utilised to create tightly fitting polygons which better represent the accessible areas within the specified distance as it wraps around the outermost points. The alpha value can be calculated, however, it is recommended to experiment for each dataset.

Due to this being a computationally expensive task, the function used **tqdm()** to print the ongoing progress to the terminal.

This function works seamlessly with the outputs of **load_osm_network()** and **nearest_node_and_name()**. Whilst it's not necessary to use those functions to prepare the data for use with **service_areas()**, it is recommended to prepare the data this way as to avoid errors and bugs as it create the correct expected data structure.

Parameters:

- nearest_node_dict (dict): A dictionary of unique name and nearest node to the graph; direct output of **nearest_node_and_name()** e.g. {'Newtownbreda Library': {'nearest_node': 11183830480}, …}.
- graph (MultiDiGraph): network graph, such as one produced by networkx; direct output of **load_osm_network()**
- search_distances (list): A list of distances in metres to search the graph by.
- alpha_value (int): Alpha value for creating the alpha shapes around furthest reachable nodes.
- weight (str|func): A string or function which assigns the weight of an edge, e.g. 'length' or 'time'. See **networkx documentation** for greater detail of options for this arg.
- progress (bool): If True, prints ongoing task, average task completion and predicted total completion time using Error! Reference source not found.
- save_output (bool): If True, saves output as service_areas.gpkg to the root folder. Defaults to False.

Example:

```
from services import network_bands

dist_list = [1000, 2000, 3000]

service_areas = service_areas(nearest_node_dict = node_dict, graph = G,
search_distances = dist_list, alpha_value = 500, weight = 'length', progress =
False, save_output = True)
```

Note: Output will result in overlapping polygons which look similar to Figure 4 (output of the example script).

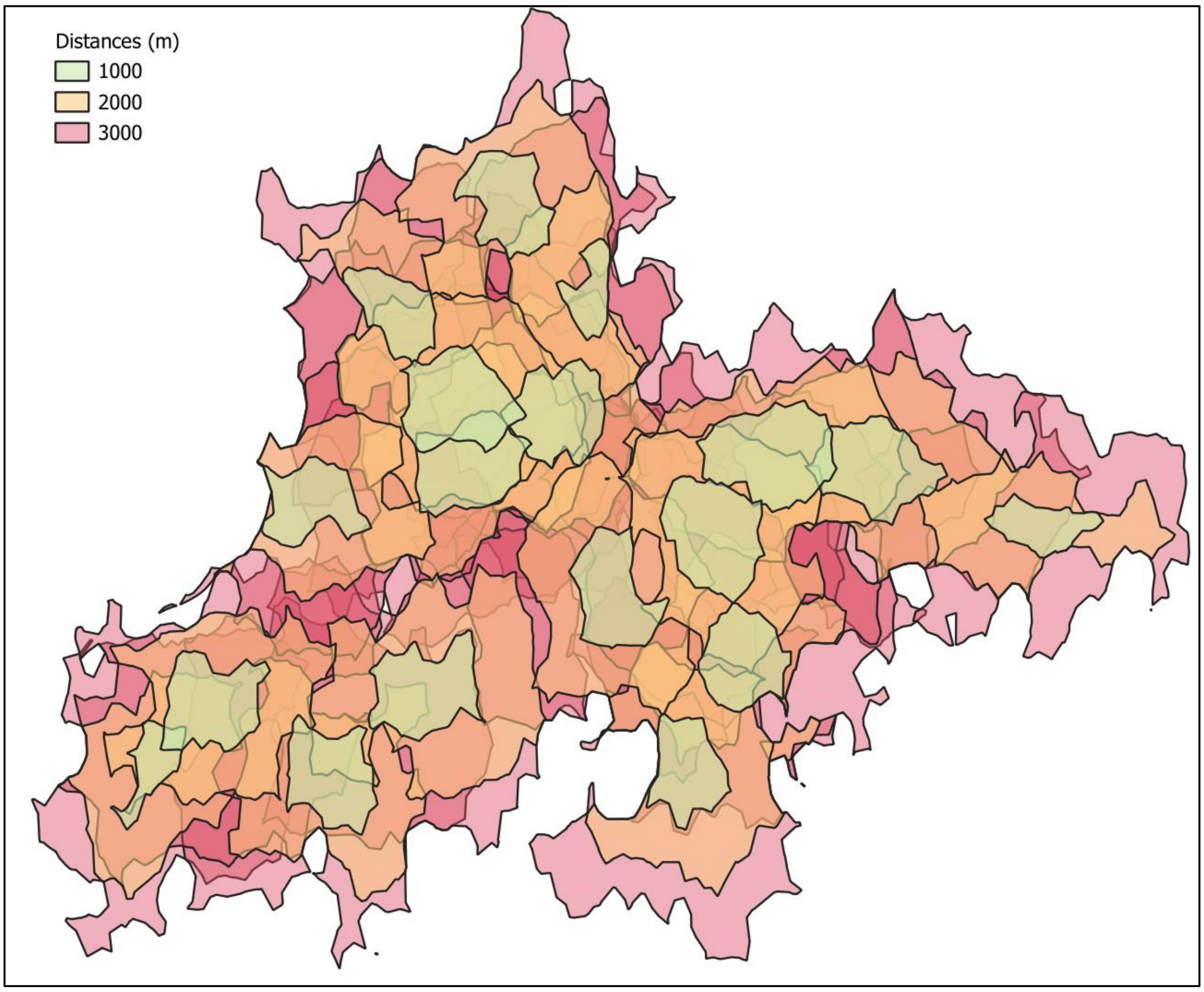


Figure 4. Example output of service_areas().

3.2.5 service_bands()

This function takes overlapping polygons, such as those produced by **service_areas()** (Figure 4. Example output of service_areas(). and dissolves the polygons into a multipart polygon by a dissolve category that **must** comprise of integers or float values and be ordered (e.g. 1000, 2000, 3000). The dissolved multipart polygons are then differenced based upon the dissolve category values, where the larger value is differenced by the subsequent lower value, iterating until the final smallest value. See Figure 5 as an example of the function using the output of **service_bands()**, where polygons with path lengths of 3000m were differenced by polygons of 2000m, and then 2000m differenced by 1000m.

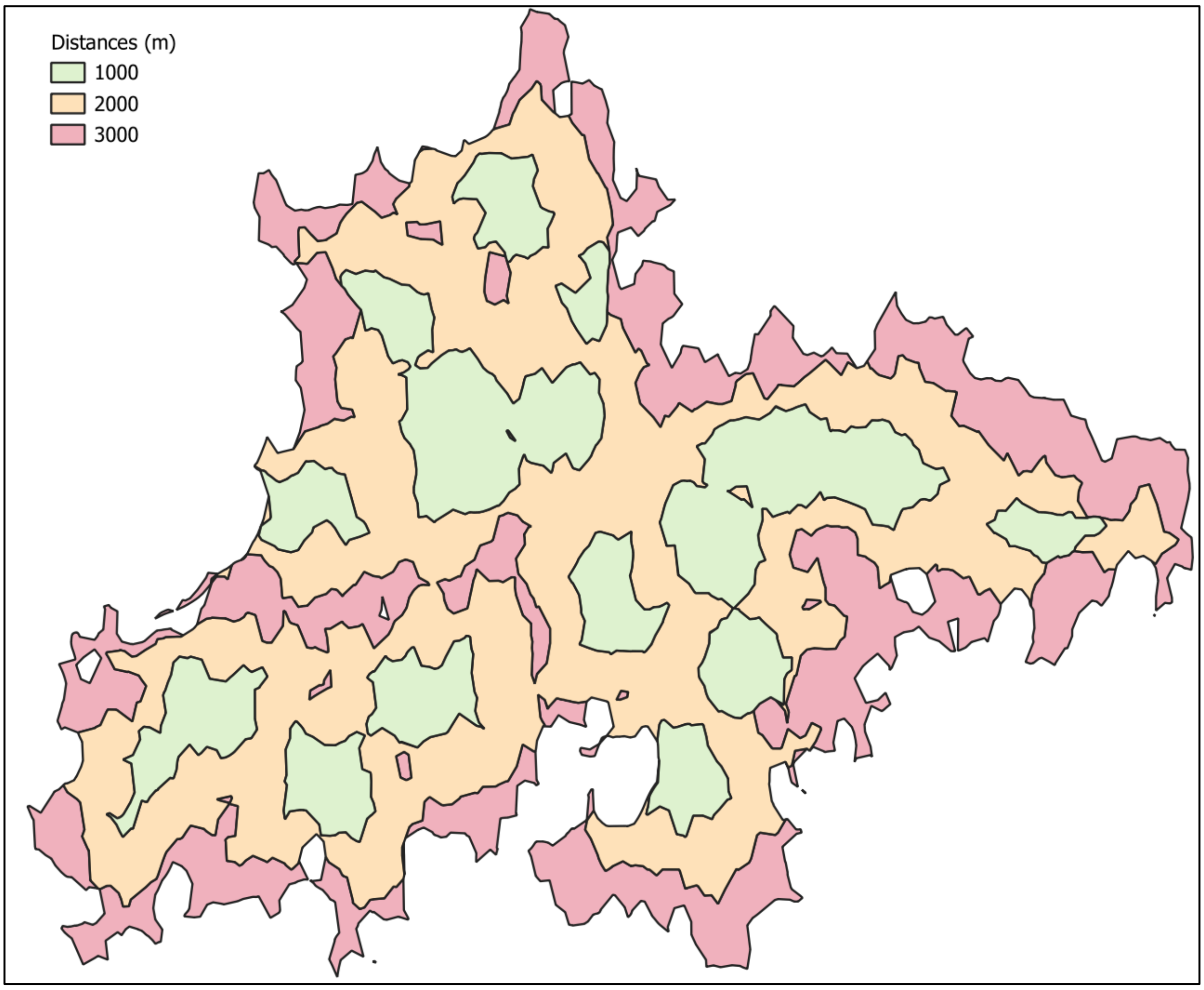


Figure 5. Output of service_bands(). Dissolved equivalent of service_areas() example output in Figure 4.

Figure 6. Output of service_bands() if show_graph = True. is the output of **service_bands()**, which has no overlapping polygons and will be plotted if the process has been successful and the argument `save_output = True`. The plot uses matplotlib with a continuous style for the polygons to ensure modularity with differing lengths of *dissolve_cat* as matplotlib does not currently natively support patches for polygons.

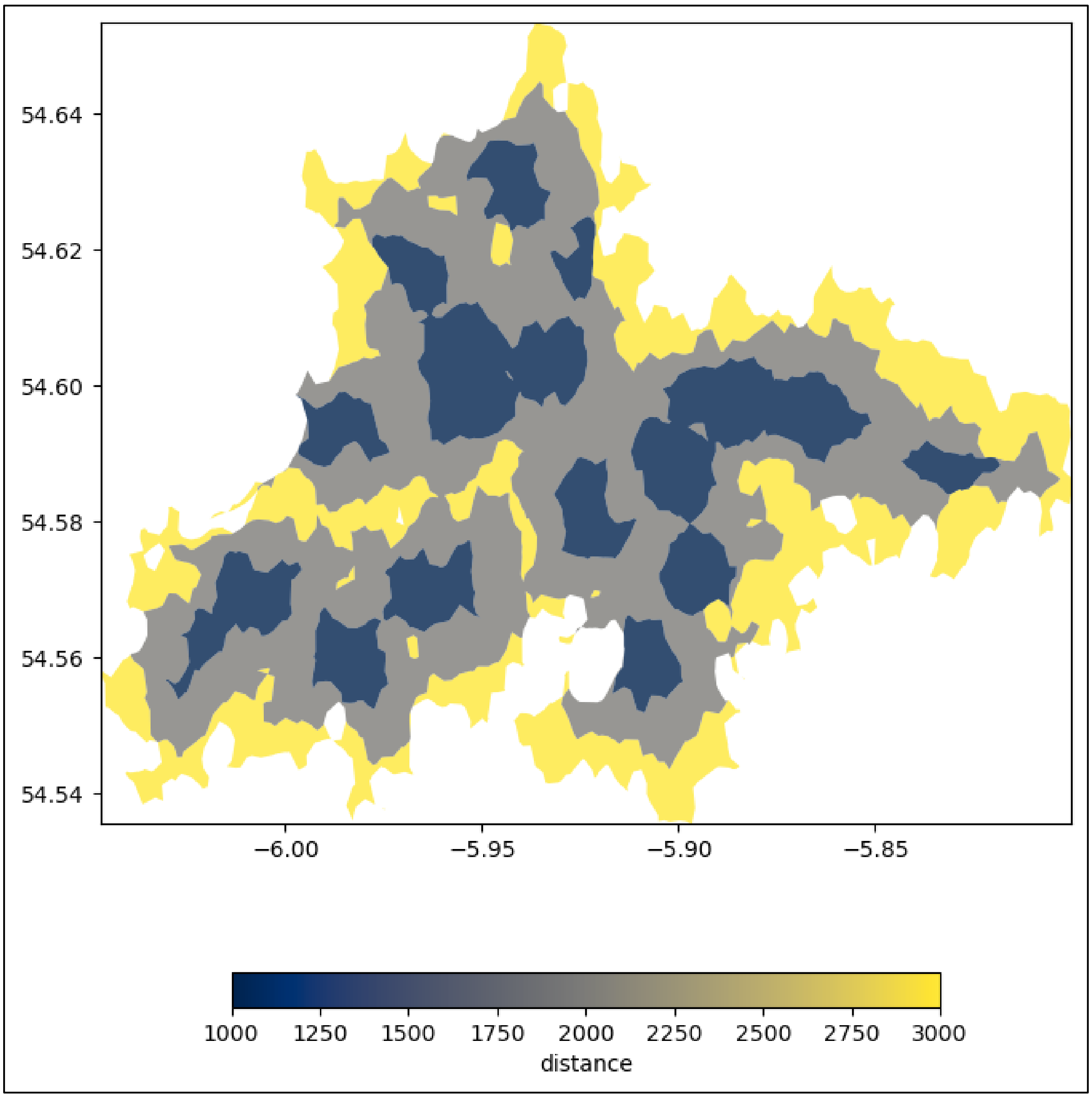


Figure 6. Output of service_bands() if show_graph = True.

Parameters:

- geodataframe (GeoDataFrame): A geopandas GeoDataFrame of polygons; can use the output of
- dissolve_cat (str): Column to dissolve the dataframe by, this column should comprise of integers by which the function orders. If using in conjunction with other
- functions, this will likely be 'distance'.
- aggfunc (str|func): Aggregation function for manipulating data associated with each group; defaults to 'first'. See [geopandas documention]() for more information.
- show_graph (bool): If true, will show a basic graph using matplotlib of output.
- save_output (bool): If True, saves output as 'service_bands.gpkg' to the root folder. Defaults to False.

Example:

```
from services import network_bands

service_bands = network_bands.service_bands(geodataframe = service_areas,
dissolve_cat = 'distance', aggfunc = 'first', show_graph = True, save_output =
True)

'A map showing network contours has been created.'
```

Note: whilst **service_bands()** has been designed for use directly with **service_areas()**, this function will work with any overlapping polygons which have a column of comprising of ordered numbers to dissolve by.

3.2.6 shortest_path_iterator()

```
shortest_path_iterator(start_locations:gpd.GeoDataFrame,
destination_locations:gpd.GeoDataFrame, networkx_graph)
```

The function calculates the nearest node on a networkx graph, utilising **nearest_node_and_name()**. It then subsequently iterates over each destination location such as a dataset of hospitals or supermarkets for each input location such as houses to calculate the distance to the nearest of the destination locations, only retaining the closest distance.

This tool is a work in progress

This function is an experimental function was initially the core of the project, however, due to inefficient computation, the calculation for realistic datasets such as >1000 locations can take hours. The example dataset of houses within Belfast for example will take 143 hours to calculate. The function will warn the user if the dataset is large and will take a long time and offer alternatives.

Ongoing development of this tool seeks to make calculating the nearest node on the graph faster. As well as this, development will seek to more intelligently calculate the nearest location as currently this is a greedy algorithm using a greedy algorithm (Dijksta's), causing excessively high computation times. This will likely be solved by incorporating service areas into the calculation to subset the data.

Example:

```
shortest_paths = shortest_path_iterator(start_locations = house_data,
destination_locations = hospitals, networkx_graph = G)
```

Parameters:

- start_locations (GeoDataFrame): geopandas DataFrame of start locations such as houses.
- destination_dataset (GeoDatFrame): geopandas DataFrame of end locations such as hospitals or supermarkets.
- network_x_graph (MultiDiGraph): graph created using networkx. Default 'G'.

3.3 batch_csv

Script containing ancillary functions for data analysis of data with the core functions which form part of the example script.

3.3.1 batch_csv_read()

```
batch_csv_read(file_paths:list)
```

Function which reads CSVs and converts them to a dictionary of pandas DataFrames. The file name becomes the accessible key and the data in the DataFrame becomes the value.

Example:

```
from services import batch_csv

# Define file paths - relative path for each csv
file_paths = [
    '/testEnvironment/Data/census_data/ni-2021-usual-residents.csv',
    '/testEnvironment/Data/census_data/ni-2021-households.csv',
    '/testEnvironment/Data/census_data/ni-2021-employment-deprivation.csv'
]

#load all defined csvs
loaded_csv = batch_csv.batch_csv_read(file_paths)
```

Parameters:

file_paths (list): A list of relative file paths to load (CSVs must be in the root folder or child folder).

3.4 census_merge

Script containing ancillary functions for data analysis of data for handling 2021 OSNI census data which forms part of the example script.

3.4.1 join_census_csv()

```
join_census_csv(dict_of_df:dict, join_column:str, drop:bool, join_type='left')
```

The function joins dataframes by common unique identifiers such as geographic codes in OSNI census data. Where there are duplicate columns names, such as geographic code, index etc, duplicates from the join_column (right join) are deleted, as such it is recommended that ‘left’ or ‘inner’ joins are used where possible. Where columns are not dropped and appended with suffixes, see **drop_dupe_cols()** to remove the duplicate columns.

If columns are dropped, the function will print: 'The following columns were duplicates from the right join and were dropped: {columns_dropped}'. If no columns are dropped, the function will print: 'No columns were dropped, all duplicate columns retained.'.

Example:

```
joined_census_data = census_merge.join_census_csv(loaded_csv, 'geography
code',  drop=True,join_type='left')

The following columns were duplicates from the right join and were dropped:
[['geography', 'access census area explorer'], []]
```

Parameters:

- dict_of_df (dict): dictionary of dataframes, a result of the mass_csv_read() function.
- join_column (str): column name to join by.
- drop (bool): Drop duplicate columns if true, else keep them.
- join_type (str): type of join - SQL-like, see pd.merge() docs. Recommended to use 'left' or 'inner' join where possible.

3.4.2 drop_dupe_cols()

```
drop_dupe_cols(df:pd.DataFrame, suffixes:tuple)
```

If after a table join or merge there are duplicated columns which are appended with suffixes such as '_left' and '_right', this function will identify which columns are duplicated and retain the left column, dropping the right column and removing the suffix. For example if the dataframe has two columns called 'name_left' and 'name_right', the 'name_right' will be dropped and 'name_left' renamed to 'name'.

Note: This **only** removes the right table join columns, it is recommended that you assign the table with columns with columns not wanting to be retained as the right table.

Example:

```
# Merge the data zones with the loaded census data.
merged_data = pd.merge(left_df, right_df, left_on = 'unique_id', right_on =
'unique_id', how = 'left', suffixes = ('_left', '_right'))

# Drop the duplicate columns from the merged dataframe
census_merge.drop_dupe_cols(df = merged_data, suffixes = ('_left', '_right'))
```

Parameters:

- df (DataFrame): Pandas DataFrame which has duplicate columns appended due to a merge or join operation, such as 'name_left' and 'name_right'.
- suffixes (tuple): Tuples (left_suffix, right_suffix) which were used to retain duplicate columns, e.g. 'name_left' and 'name_right'.

pandas_aux

Script containing ancillary functions for data analysis which extend the usage of pandas.

3.4.3 fill_na_with_zero()

```
fill_na_with_zero(df:pd.DataFrame, columns:str)
```

Replaces all NaN values with a ‘0’ in specified columns in a pandas DataFrame.

Example:

```
from services import pandas_aux as pdaux

#Replace all NaN values in belfast_zones_census with 0
pdaux.fill_na_with_zero(belfast_zones_census,
['households_1000','households_2000','households_3000'])
```

Parameters:

- df (DataFrame): Pandas DataFrame with columns to modify.
- columns (list): List of column names to fill NaN values in.

3.4.4 append_col_prefix()

```
append_col_prefix(df:pd.DataFrame, col_names:list, prefix)
```

Append a prefix to a column in specified columns in a pandas DataFrame. Useful for when grouped data is unstacked, for example a DataFrame with column names such as 1000.0, 2000.0 will be converted to prefix_1000, prefix_2000.

Example:

```
df.column # dataframe with numerical values
Index([1000.0, 2000.0, 3000.0], dtype='float64', name='distance')

df_renamed = pdaux.append_col_prefix(df, col_names = [1000,
2000,3000],  prefix = 'households')

# Print renamed columns.
df_renamed.columns
Index([households_1000, households_2000, households_3000], dtype='object)
```

Parameters:

- df (DataFrame): input dataframe, can be geopandas or pandas.
- col_names (list): List of columns to append.
- prefix (str or int): Prefix to append to the column names.

4 Troubleshooting

4.1 Common Issues:

- Problem: Module/package not found.
- Solution: Ensure packages are installed, imported and the using the correct python interpreter. If running the example, ensure netgeo_env is activated.
  Python 3.9, 3.10, 3.11 and 3.10 only have official support, older versions will not work.
- Problem: DriverError: {file} does not exist in the file system.
- Solution: Ensure that data is stored within the root folder or below and not elsewhere.

4.2 Data Loading Issues:

- Problem: Errors in loading shapefiles or CSV files.
- Solution: Check the file paths, ensure the CSV / other data are not corrupted, and validate the structure (headers, formats etc).
- Problem: Data structure.
- Solution: Ensure that the data structure fits the function, for example, **from services import network**_bands

```
roads = ‘C:/Users/user/projects/Service-Area-Tools/belfast.osm.pbf’

G, nodes, edges = network_bands.load_osm_network(file_path = roads, >>>
network_type = 'driving', graph_type = 'networkx')
```

- csv_to_gdf() requires separate coordinate columns for X and Y and at least a specified input CRS.

4.3 nearest_node_and_name() errors:

- Problem: Dictionary of nearest nodes has random city names.
- Solution: Ensure that the arg `anon_name` is `False`

4.4 service_areas() errors:

- Problem: No output (polygons or network_areas.gpkg) generated when using service_areas().
- Solution: Ensure input locations are set to the correct CRS, if not already in EPSG: 4326, reproject accordingly.
  If expecting a geopackage file, ensure the ‘save_output = True’ as part of the arguments.
- Problem: error with search_distances.
- Solution: Ensure the search_distances comprise of integers or float values only.

4.5 service_bands() errors:

- Dissolving error.
- Ensure that the values within the 'dissolve_cat' are all integers or floats, if there are NaN values, convert these prior to running the function.
- Problem: Output is showing polygons in random corners of the map.
- Solution: Ensure that the CRS of input locations to **nearest_node_and_name()** and subsequently **service_areas()** are all in EPSG: 4326 if using OSM data for the network graph.

4.6 CRS Errors:

- Problem: issues with CRS transformations, spatial join errors.
- Solution: Verify the CRS of all datasets have been assigned the correct EPSG code.

4.7 Network graph errors:

- Problem: network graph does not get created from OSM data.
- Solution: Ensure the data is a **.pbf** file, these can be downloaded from **osmconvert** or **Osmosis**.

  If you are unsure if the graph has been created, you can plot the graph simply with matplotlib with `edges.plot()`, which will produce an output of the roads if it the graph has successfully been generated.

5 References

Comber, A.J., Brunsdon, C. and Radburn, R. (2011). A spatial analysis of variations in health access: linking geographi, socio-economic status and access perceptions. *International Journal of Health Geographics*. 10(44).

Foley, R. and Darby, N. (2002). Placing the practice: GP service location planning using GIS in Brighton & Hove. *Geohealth*.

OpenStreetMap (2014). OpenStreetMap Data. https://www.openstreetmap.org/

QGIS.org (2024). QGIS Geographic Information System. QGIS Association. http://www.qgis.org

Suárez-Vega, R., Santos- Peñate, D. R. and Dorta-González, P. (2012). Location models and GIS tools for retail site location. *Applied Geography*. 35(1-2), 12-22.