BGC Explorer

Key

For convenience, the following text formatting convention is used in this document:
`This indicates a file location or path`
`This is something you should type within python (following >>>)`
`This is something you should type in command line window (following >)`
This is a web link

About

The Biosynthetic Gene Cluster Explorer (BGC Explorer) is a python module written by Clint Cario as a rotation project for the Fischbach lab. It uses bacterial organism, gene, and domain functional data from the JGI IMG database and predicted gene clusters (determined by Peter Cimermancic's Clusterer algorithm) to group BGCs into families based on gene set similarity and filtering criterion (eg. taxa and BGC product). It also provides a way of visually exploring these clusters to find patterns and similarity.

More information can be found in a reveal.js presentation found in the code repository: `Rotation Presentation/index.html.` Simply open this file in your favorite web browser to view it.

Installation

Python Prerequisites

The BGExplorer was written and tested in python 2.7, so you will need to download and install that for your specific OS: https://www.python.org/download/

If you have version python 3.x, you are welcome to try the code with it, but functionality is not supported.

In addition to python, you will need to install several modules (or packages/eggs), which extent the functionality of python and the BCGE. There are 3 ways to do this:

1. Download the module code directly from the module's website and compile it, or if a `setup.py` file is present, by running `> python setup.py` on a command line in the directory of the extracted code.

2. Use your OS's package management system to get the libraries. Many times, the libraries are prefixed with 'python', like `python-pewee`. Examples package management systems include:
    a. MacOSx
        i. Fink
        ii. Homebrew
        iii. MacPorts
    b. Linux
        i. apt-get
        ii. yum
    c. Windows
        i. .msi files

3. Use a package manager specific to python (**recommended**):
    a. setuptools (easy_install)
        i. Follow instructions for your specific OS, which usually includes:
            1. Downloading `ez_setup.py`
            2. Running it with python:
                a. `> python ez_setup.py` where you downloaded the file
            3. This should install a tool called easy_install, which works like
                a. `> easy_install module_name`
    b. pip (**recommended**)
        i. Follow instructions on the website.
            1. Download `get-pip.py`
            2. Run it with python:
                a. `> python get-pip.py` where you downloaded the file
            3. You should now be able to install new modules using:
                a. `> pip install module_name`

Notes:

- Some packages require root (administrator) privileges:
    a. In Unix, Linux, and Mac, prefix the install command with `sudo`
    b. In Windows, prefix with `runas /noprofile /user:Administrator`
- You may need to try several of the above methods to get the modules to install
    a. Sorry, this is just how python is sometimes.
    b. I hate this too!
- If you are comfortable with Linux, the easiest way to accomplish all these things is with Ubuntu and `> sudo apt-get install python-module_name`. This is very reliable.

So, which modules do I need?

Preinstalled

The following modules are required but should have been pre-installed with python. These are safe to skip for now, but if python starts complaining about needing one of them, try to install it as you would any other module (see above).

- os
- sys
- subprocess
- tempfile
- threading
- shutil
- math
- json
- webbrowser
- SimpleHTTPServer
- SocketServer

Required

The following are required and should be installed (using easy_install, pip, or OS specific package manager, as above):

- numpy (follow instructions on the site)
- munkres
- peewee (recommends pip or manual install)
- Bio (Biopython; recommends easy_install)

Optional

These are not required, but are super useful for debugging:

- pprint

BCExplorer Code

Congrats, you've made it past the hardest part. All you have to do now is make sure you get a copy of the BGC Explorer code!

There are four ways to do this:

1. Clone the code repository with git (the most recent version of the code inc. bug fixes)
    a. `> git clone https://github.com/lucidv01d/BGCexplorer.git`

   b. or, if you have already done step a: `> git pull` (in the code directory) to get the most recent version of the code
   c. Make sure a copy of the database BGCs.db exists in the data/ directory
2. Dropbox
   a. The dropbox should contain all files, including the raw files in the data/ directory (which were used to generate the database).
3. USB stick (**recommended & easiest**)
   a. I will provide a hard copy of the final code on a small USB stick (given to Michael).
4. Ask me (clint.cario@gmail.com)

Essential

Make sure you got these at least these files:

- BGCExplorer.py (the main script)
- BGC_models.py (the database definition file for peewee)
- Arrower.py (the BGC arrow generating script)

Quasi-essential

These should both be included, but you can get by with just one of them.

- BGC_Databaser.py and complete data/ directory (Human*/ subfolders, etc...)
- data/BGCs.db (the database, which is generated by BGC_Databaser.py)

I don't have the database data/BGCs.db or I need to recreate it with new data

You're in luck. If you have the flat files from JGI and Clusterer, there is a script that will build the database for you. Please refer to the reveal.js presentation for the format of the raw files and make sure you have the following:

1. An BGC annotation file (something like data/BGC_annotation_final.out)
2. An organism table (eg. data/AllOrgsTable.out)
3. Clustered BGC files (eg. data/HumanBGCs/*clusters.out and data/HumanBGCs/*.out)
4. BGC Sequence files (eg. data/HumanBGC_seqs/*.fasta)

Next, make sure the following lines in BGC_databaser.py reflect where these files are located: [Lines 16-20]

```
DATA_DIR       = os.path.abspath(os.path.join(cwd, os.pardir, "data"))
ANNOT_FILE     = os.path.join(DATA_DIR, "BGC_annotation_final.out")
ORGS_FILE      = os.path.join(DATA_DIR, "AllOrgsTable.out")
BGCS_DIR       = os.path.join(DATA_DIR, "HumanBGCs")
BGCS_SEQS_DIR  = os.path.join(DATA_DIR, "HumanBGCs_seqs")
```

Now run the database populator:

- `> python BGC_databaser.py`

If you need to reset the database (update an existing version):

- `> python BGC_databaser.py reset`

You should now start a python interpreter in the directory with the code and type:

```
>>> from BGCExplorer import *
>>> BGCs.precompute_similarities()
```

This will compute pairwise similarity between all domains in the database, which will save tremendous time in subsequent analyses.

Usage

You are now ready to start using the code. Unless otherwise specified, all commands below are run in the directory containing the code or in a python interpreter.

There are three ways to interact with the code:

1. Import the library into an interactive python shell and use it
2. Generate a script that uses the library
3. Use an IDE like spyder2, which is a combination of 1&2

The only difference between these is whether you want to save a set commands to run repeatedly with little interaction (2) or if you prefer to interface with the code interactively (1 & 3). In any case, the sequence of events is the same (these can be found in the `sample_script.py` file)

1. Import the library
   ```
   >>> from BGCExplorer import *
   ```
2. Create a new named filter (refer to filters section of the presentation: Rotation Presentation/index.html#/9/4)
   ```
   >>> NRPS = BGCs(filters={'Kind':"NRPS", 'genus':'Bacteriodes'})
   ```
3. Do a pairwise similarity comparison of all BGCs that have been filtered from from the database. This operation will store results back to the database for much faster subsequent lookup. Therefore, once this operation has completed successfully for a given filter, it doesn't have to be done again.
   ```
   >>> NRPS.compare_bgcs()
   ```
4. Next, we can query NCBI for additional information on our selected BGCs. This information includes functional annotation and gene orientation. Again, these results are stored back to the database, so this operation only has to be performed once per filter.

```
>>> NRPS.augment()
```

5. Next, we should cluster the BGCs, which groups them into families based on their pairwise similarity scores. This is done by pruning similarity scores (edges) between BGCs (nodes) that don't reach a set threshold (default of .35 [range 0-1]). Additionally, the mcl algorithm can be used to threshold, or both. Please see function definitions in the presentation for more information: Rotation Presentation/index.html#/9/6
   ```
   >>> NRPS.cluster(cutoff=.7)
   ```
6. Now we can visualize, either using a network visualization or a bubble chart. The network visualization shows BGCs as nodes in a graph with similarity score edges connecting them. Only edges that have reached the threshold are shown, and this allows us to determine BGC families. Nodes are additionally color coded by an annotation field which can be selected in a dropdown dialog box.
   ```
   >>> NRPS.visualize_network()
   ```

   Another visualization feature is the cluster bubble chart. This representation is similar to the network visualization above, with the exception that connected nodes are grouped together into a bubble with a radius proportional to the number of nodes. These are then differentially colored to represent different groups. Bubbles can be clicked on to show arrower diagrams which represent BGC gene structure of each BGC of a group. Clicking on the BGC will bring up information about it into a 'card' on the screen, and clicking anywhere else will close the arrower diagram overlay.
   ```
   >>> NRPS.visualize_clusters()
   ```

Notes:

- If you do multiple visualizations, you may have to force your browser to reload data structures by pressing Ctr+Alt/Apple+R.
- There is no programmatic limit on the number of BGCs that can be visualized, but a network/cluster with more than 1,000 BGCs is likely to have very poor performance.
- Likewise more than 1,000 edges will also cause performance issues.
- The 'verbose' mode (default = True) of the cluster() function will tell you the number of nodes/edges that you have selected.

Advanced Usage

Initializing the filter object (step 2 above) sets parameters that generate a SQL query that is used to filter the database. The query is generated by the instance's `db.build()` method (eg. `NRPS.db.build()`), which is then stored in `db.query`. If you require more complex filters like boolean logic ones using OR, AND, or NOT, then you can specify your own SQL query to filter the database.

To do this, simply replace the `db.query` string with your own custom SQL query. You can also build off of the one already generated:

```
>>> NRPS.db.query = '''
>>>   SELECT * FROM bgc
>>>   INNER JOIN organism ON organism.organism_id = bgc.organism_id
>>>   WHERE bgc.kind LIKE 'saccharide'
>>>   AND bgc.kind NOT LIKE '%poly%'
>>>   AND organism._genus LIKE 'Bacteroides'
>>> '''
```

Information on the SQL query language can be found online.

You can also use your object to access dictionaries containing BGCs, genes, or domains:

```
>>> NRPS.db.get_genes()
>>> NRPS.db.get_bgcs()
>>> NRPS.db.get_domains()
```

and also clustered or networked nodes/edges:

```
network:
  {
    'nodes':  [{ 'name' => bgc_id, 'attribs' => bgc }, ... ],
    'edges':  [{ 'source' => node idx, 'target' => node idx, 'value': edge_weight }, ...
],
    'mapping': { bgc_id => node_idx }
  }

clusters:
  {
    'nodes': [{ 'cluster' => group_idx, 'attribs' => bgc, 'radius' => 10}, ... ],
    'groups':[ [{bgc_id => bgc}] ],
    'mapping': [ {bgc_id => group_idx} ]
  }
```