ChemTreeMap Documentation

Release 0.1.0

Jing Lu

July 30, 2016

Contents

CHAPTER 1

Contents

1.1 Introduction

ChemTreeMap is a novel program for visualization of biochemical similarity in molecular datasets. It allows for users to interactively explore the chemical space covered by a set of small molecules. It is written by Jing Lu, a PhD student under the supervision of Dr. Heather Carlson.

1.1.1 Software

Briefly, the data processing is done using Python for calculations of similarity, and JavaScript for visualization:

The workflow is shown in the following Fig. 1.1.

There is a demo (http://ajing.github.io/ChemTreeMap/index.html#example) of the interactive frontend for some example datasets hosted on GitHub.

1.1.2 Paper

ChemTreeMap: An Interactive Map of Biochemical Similarity in Molecular Datasets.

The paper has been submitted to Bioinformatics.

1.2 Installation

ChemTreeMap contains two parts: frontend (JavaScript) and backend (Python). Two methods of installation are supported.

- Docker installation: Run ChemTreeMap in a Docker container with a reproducible environment.
- Source code installation: The current source code installation instruction is only tested for Ubuntu (Linux) system.

the dependency rapidnj (http://birc.au.dk/software/rapidnj/) and graphviz (http://www.graphviz.org/) do not have command line support for Windows. Please check this wiki page (https://github.com/ajing/ChemTreeMap/wiki/Souce-code-installation) for more details. The ChemTreeMap backend can be installed as a python package.

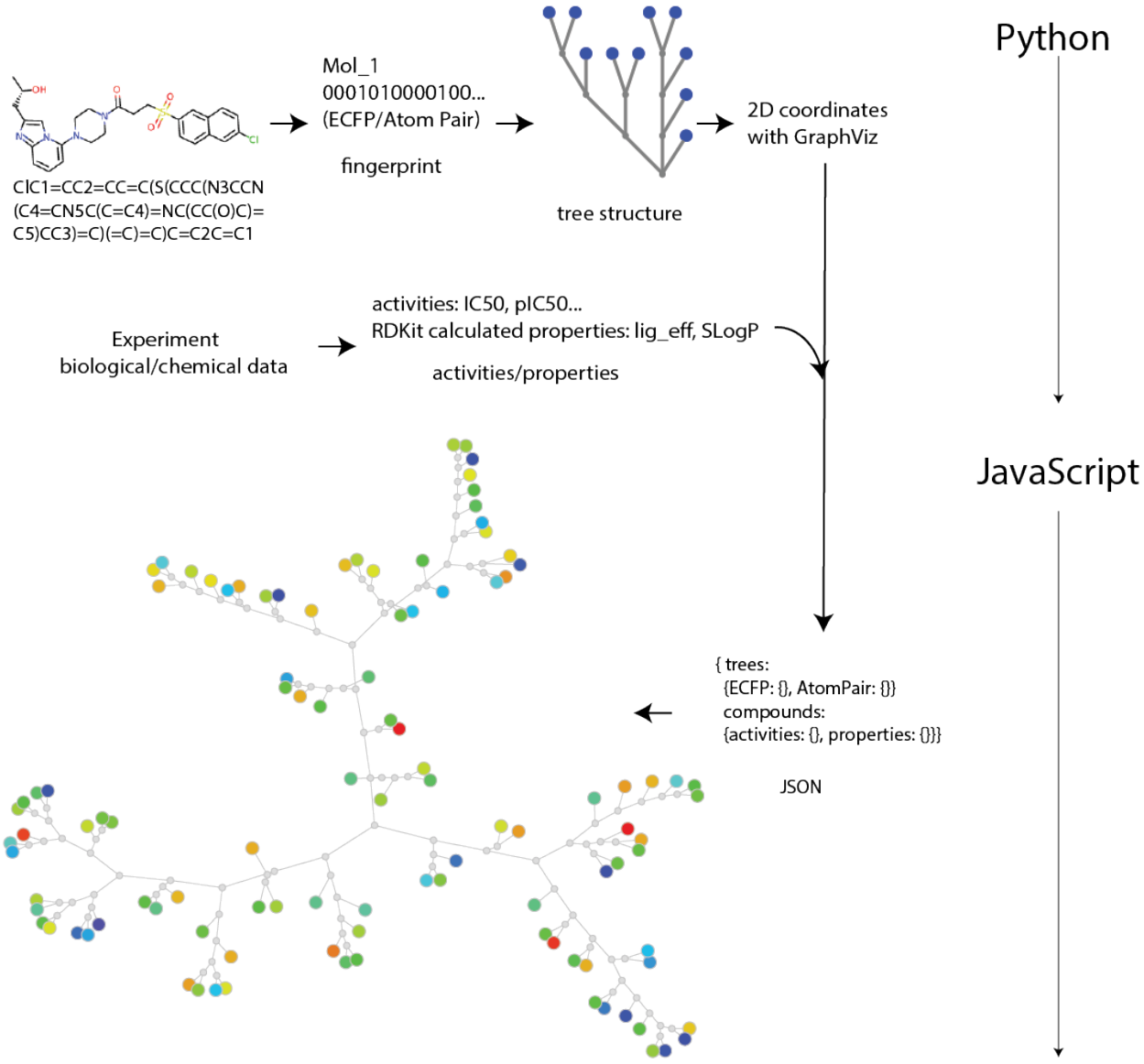

Fig. 1.1: The workflow for ChemTreeMap

1.2.1 Docker installation (fastest install)

Docker provides a reproducible environment for scientific software. All dependencies are pre-installed in the Docker container.

Docker is a system which can be used to build self contained versions of a Linux operating system running on your machine. When you install and run ChemTreeMap via Docker, it completely isolates the installation from pre-existing packages on your machine.

Please follow the following steps to install the ChemTreeMap Docker container. (Superuser privilege may be necessary for the following commands)

See installing Docker (https://docs.docker.com/engine/installation/) for instructions on installing Docker on your machine. Note: VirtualBox (https://www.virtualbox.org) must be installed to run Docker.

Installing Docker on Windows:

https://docs.docker.com/engine/installation/windows/

Installing Docker on Mac:

https://docs.docker.com/engine/installation/mac/

Installing Docker on Ubuntu:

https://docs.docker.com/engine/installation/linux/ubuntulinux/

After Docker is installed, pull ChemTreeMap image using the following commands.

```
$ docker pull ajing/chemtreemap
```

Then, launch a Docker container with the binary image as follows.

```
$ docker run -t -i -p 8000:8000 ajing/chemtreemap /bin/bash
```

The example code is in ~/examples folder. Please read and run the examples.py file to see how it works with example input files.

```
$ cd examples
$ python examples.py
```

Then, open the following URL in Chrome/Firefox web browser. The following URL is for Linux Systems (Mac, Ubuntu, Fedora ...).

http://localhost:8000/dist/#/aff

'aff' may be changed to your input filename without file extension.

In using a Windows system, the docker daemon is running inside a Linux virtual machine. So, you need to use the IP address of your Linux virtual machine. It is shown in your Docker QuickStart Terminal (Figure 1.2) in the red box:

Fig. 1.2: The IP address (red box) on Docker QuickStart Terminal.

With the IP address, open the visualization with the following URL in your Chrome/Firefox browser. In this example, use

http://192.168.99.100:8000/dist/#/aff

If you have your own molecule files, please prepare your input file following the same tab delimited format as example files (e.g. aff.txt, factorxa.txt, etc.). The following command can be used to import files into Docker container.

```
$ docker cp foo.txt ajing/chemtreemap:/examples/foo.txt
```

Change examples.py (variables input_file, out_file) accordingly and run the examples.py. (Note: you may also need to change the column header to match.)

Now, you are done with ChemTreeMap installation.

1.2.2 Source Code Installation

The following sections (1.3 and 1.4) is for installing from source code (Frontend and Backend). The procedure has been tested on Ubuntu 14.04.

First, pull the source code from the GitHub repository.

```
$ git pull https://github.com/ajing/ChemTreeMap
```

1.3 Frontend

The frontend, written in JavaScript, requires modern (HTML5 enabled) web browsers. It is an angularjs app, which was scaffolded using yeoman, and angular generator. It uses the D3.js library to render images and the force directed graphs.

1.3.1 Dependencies

The project requires the node (https://nodejs.org/en/) runtime to build the app, Grunt (http://gruntjs.com/) to run tasks, and the package managers npm (https://www.npmjs.com/) and bower (http://bower.io/) to install development and production dependencies respectively.

1.3.2 Building

To build and serve the project (having installed node and npm), you can change to the frontend directory and run:

```
$ npm install -g grunt bower      // install grunt and bower
$ npm install                     // install node modules
$ bower install                   // install bower modules
$ grunt build                     // build the server
$ python -m SimpleHTTPServer 8000 // run the server
```

Your default browser should open by default to `localhost:8000`, where the app is being served.

1.4 Backend

The backend is written in *Python*, and is where data is processed into the JSON representation. The API is detailed in treebuild.

1.4.1 Installation

First, install rdkit (http://www.rdkit.org/), ete (http://etetoolkit.org/), and graphviz (http://www.graphviz.org).

Switch to the backend directory and run the *setup.py* script.

```
$ python setup.py install
```

1.4.2 Quickstart

A prebuilt script, `examples.py` for analysing composite biochemical data in a standard format is provided in the `examples` folder. This provides minimum functionality - the software is designed for extensibility, so consider writing your own scripts like those in the example datasets included in `backend/treebuild/examples/`.

The script can be run using:

```
$ python examples.py
```

The server will serve the output upon running `examples.py`

Data structure

Compounds

The example script takes compounds as a smiles file. The smiles should be provided with column name Canonical_Smiles, and the identifier column is set as a parameter of TreeBuild object. The individual activities and properties are supplied as extra columns.

A sample of a compound dataset would be delimited by tabs:

```
Canonical_Smiles        ligandid        pIC50
COc1cc2Cc3c(n[nH]c3c4ccc(nc4)c5ccc(O)cc5)c2cc1OC        Chk1N144        9.52432881
COc1cc2Cc3c(n[nH]c3c4ccc(nc4)C#N)c2cc1OC        Chk1N145        9.09151498
OC1CCCN(C1)c2ccc(Br)cc2NC(=O)Nc3cnc(cn3)C#N     Chk1N40 8.86646109
COc1c(OC)cc2c(Cc3c2n[nH]c3c4ccc(c5ccc(O)cc5)cc4)c1      Chk1N115        8.79588002
... ... ...
```

Showing 4 compounds, with pIC50 as activity.

Running the Script and Viewing the Tree

The script will use the default fingerprint (ECFP and AtomPair) and caculated properties (pIC50, SlogP and ligand efficiency) to generate the ChemTreeMap data, and produce JSON output file, specified in the out_file parameter of TreeBuild object.

```
$ python examples.py
... ... ...
```

The generated data will be copied to the data directory of the Frontend and viewed in a browser.

Then open the browser Chrome/Firefox with the link (http://localhost:8000/dist/index.html#/aff).

aff needs to be changed to the filename of your input file without file extention.

Please consider looking at `treebuild/examples/examples.py` where there are five examples using the declarative API for generating the datasets.

CHAPTER 2

Modules

2.1 treebuild

2.1.1 treebuild package

Submodules

treebuild.tree_build module

class `treebuild.tree_build.`**`TreeBuild`**(*input_file*, *output_file*, *id_column*, *fps*, *properties*)

There are assumptions for the data format of the input file. It is very important to understand these assumptions:

1.potency (e.g. IC50/Ka/Ki) unit is nM

2.the file must have a id column, you can set the column name with id_column

3.the file must have a SMILES column, with 'Canonical_Smiles' as column name

4.the file must have at least one potency column (IC50/Ka/Ki).

To build the tree

1. the identity column needs to be specified with id_column
2. a list of fingerprints and a list of properties need to be specified with rdkit
3. the directories for input and output file need to be specified

static `dot2dict`(*dot_outfile*)

static `gen_dist_file`(*liganddict*, *fp_func*)

generate distance file which is the input of rapidnj program.

Parameters

- **`liganddict`** – ligand information
- **`fp_func`** – fingerprint function

Returns filename for distance file

static `gen_properties`(*ligand_dict*, *activities*, *properties*, *ext_cols*)

Generate properties for each molecule.

Parameters

- **ligand_dict** – ligand dictionary which keep all ligand information
- **activities** – a list of PropertyType objects
- **properties** – a list of PropertyType objects
- **ext_cols** – the column name for external links

Returns

static make_structures_for_smiles(*ligand_dict*)
Make structure figures from smile strings. All image files will be in the IMG_DIR

Parameters **ligand_dict** – ligand dictionary which keep all ligand information

Returns

static parse_lig_file(*in_file*, *identifier*)
parse ligand file and return a dictionary with identifier as IDs

Parameters

- **in_file** – input file directory
- **identifier** – name for the identifier

Returns a dictionray with ligand information

run_rapidnj(*distance_file*)
run rapidnj program on distance_file

Parameters **distance_file** – directory of distance file

Returns newick string

sfdp_dot(*dot_infile*, *size*)
run sdfp on dot file

Parameters

- **dot_infile** – directory for dot file
- **size** – parameter for the sfdp

Returns new filename

static write_dotfile(*newick*)
write newick string as dot file

Parameters **newick** – newick string

Returns dot file

treebuild.types module

class treebuild.types.**FingerPrintType**(*name*, *fp_func*, *metadata*)
representing fingerprint types

to_dict()
Show the information for this fingerprint

Returns dictionary with basic info

class treebuild.types.**PropertyType**(*name*, *metadata*, *transfunc=None*, *colname=None*)
representing biological or chemical properties

gen_property(*mol_dict=None*)
generate value for this property type

Parameters

- **prop_name** – the name of the property
- **mol_dict** – other information about the molecule

Returns a generated value for this property

set_col_name(*col_name*)
Set the property name from the input file

Parameters **col_name** – original column name in the input file

Returns

to_dict()
Show the information

Returns dictionary with basic info

treebuild.util module

Provide utility functions

treebuild.util.**AddNewChild**(*contents*, *a_node*, *new_node_name*, *edge_length*, *children*, *currentlist*)
Add a new child to a node.

Parameters

- **contents** – a string, a line from DOT
- **a_node** – a node object
- **new_node_name** – new node name
- **edge_length** – the length of edge
- **children** – existing children
- **currentlist** – current list of node name

Returns void

treebuild.util.**CleanAttribute**(*attr*)
Clean attribute, remove ‘,’.

Parameters **attr** – old attribute string

Returns new string

treebuild.util.**ConvertToFloat**(*line*, *colnam_list*)
Convert some columns (in colnam_list) to float, and round by 3 decimal.

Parameters

- **line** – a dictionary from DictReader.
- **colnam_list** – float columns

Returns a new dictionary

treebuild.util.**Dot2Dict**(*dotfile*, *moldict*)
Read a DOT file to generate a tree and save it to a dictionary.

Parameters

- **dotfile** – DOT file name
- **moldict** – a dictionary with ligand information

Returns a dictionary with the tree

treebuild.util.**GetAttributeValue**(*attrname*, *attr*)
Get node attribute.

Parameters

- **attrname** – name of the attribute
- **attr** – the attribute string

Returns the value for the attribute

treebuild.util.**GetNodeProperty**(*line*)
Get node property from a string.

Parameters **line** – a string

Returns name, size, and position of the node

treebuild.util.**GetRoot**(*dotfile*, *rootname*)
Return root name with rootname.

Parameters

- **dotfile** – DOT file
- **rootname** – the name of the root

Returns the object of the root

treebuild.util.**GetSize**(*width*)
Get the size.

Parameters **width** –

Returns

treebuild.util.**GuessByFirstLine**(*firstline*)
Guess the number of columns with floats by the first line of the file

Parameters **firstline** –

Returns

treebuild.util.**IsEdge**(*line*)
Whether this line in DOT file is an edge.

Parameters **line** – a string line in DOT file

Returns True or False

treebuild.util.**NameAndAttribute**(*line*)
Split name and attribute.

Parameters **line** – DOT file name

Returns name string and attribute string

class treebuild.util.**Node**(*name*, ***attr*)
Bases: dict

class for node of tree, each node can only have one parent

add_child(*a_node*)
Add child to the node.

Parameters **a_node** – Node object

Returns void

get_dist(*a_node*)
get the node as a dictionary.

Parameters **a_node** – Node object

Returns a dictionary

set_dist(*dist*)
set the dictionary attribute for the Node object.

Parameters **dist** –

Returns

set_parent(*a_node*)
Set the parent for a node.

Parameters **a_node** – Node object

Returns void

treebuild.util.**NodeByName**(*name*, *contents*)
Create node with name name.

Parameters

- **name** – a string with node name
- **contents** – a list of string from DOT file

Returns node object

treebuild.util.**NodeNameExist**(*line*)
Functions for parsing DOT file.

Parameters **line** – a line from DOT file

Returns whether there is a node name in this line

treebuild.util.**ParseLigandFile**(*infile*, *identifier*)
Parse ligand file to an dictionary, key is ligand id and value is a dictionary with properties and property values. This program will guess the type for each column based on the first row. The program will assume there is only two types of data: number and string.

Parameters

- **infile** – input filename
- **identifier** – the identifier column name

Returns a dictionary

treebuild.util.**ProcessName**(*name*, *isedge*)
Process the name of the node.

Parameters

- **name** – name of the node
- **isedge** – whether this is a edge

Returns new name

treebuild.util.**RecursiveNode2Dict**(*node*, *info_dict*)
Recursively populate information to the tree object with info_dict.

Parameters

- **node** – tree object with all info
- **info_dict** – information for each ligand.

Returns a tree dictionary

treebuild.util.**RemoveBackSlash**(*dotfile*)
Rewrite dot file, with removing back slash of dot file.

Parameters **dotfile** – DOT file name

Returns void

treebuild.util.**SelectColumn**(*lig_dict*, *colname*)
Prune the dictionary, only attribute in colname will be left.

Parameters

- **lig_dict** – a tree like dictionary
- **colname** – what attribute you want to keep.

Returns a new dictionary

treebuild.util.**SizeScale**(*size*)
Rescale the size (currently only convert to float).

Parameters **size** – a string

Returns a float

treebuild.util.**ToFPObj**(*alist*, *fp_func*)
A list of SMILE string object with (id, smiles) to a list of fingerprint object with (id, fp_obj)

Parameters

- **alist** – two element list, the first item is ligand name, the second is smile
- **fp_func** – the fingerprint function

Returns two element list, with first item as ligand name, second item as a fingerprint object.

treebuild.util.**WriteAsPHYLIPFormat**(*smile_list*, *fp_func*)
Prepare the input for RapidNJ.

Parameters

- **smile_list** – a list of smiles string
- **fp_func** – the fingerprint function

Returns the filename with PHYLIP format (input for rapidnj)

treebuild.util.**WriteDotFile**(*newick*)
Write newick string to a DOT file

Parameters **newick** – a string with newick tree structure

Returns DOT file name

treebuild.util.**WriteJSON**(*dict_obj*, *outfile*, *write_type*)
Dump json object to a file.

Parameters

- **dict_obj** – dictionary object
- **outfile** – output file name
- **write_type** – append or rewrite ('a' or 'w')

Returns void

treebuild.util.**extendChildren**(*a_node*, *contents*, *cur_list*)

Find all children of a node in a tree.

Parameters

- **a_node** – a node in a tree
- **contents** – contents from DOT file
- **cur_list** – current children

Returns a list of node objects (children)

treebuild.util.**getSimilarity**(*fp1*, *fp2*)

Generate similarity score for two smiles strings.

Parameters

- **fp1** – fingerprint object (rdkit)
- **fp2** – fingerprint object (rdkit)

Returns Tanimoto similarity

t