GWHQ python cookbook

13 June 2018

Intro

required:
- text expansion/shortcut software
- anaconda (python, ipython)
- pycharm

conventions:
4 spaces = tab
d is dataframe / data
h is header / metadata

scope:
csv, gct manipulation + analysis

keyboard shortcuts:
F6 : copy currently selected file path to clipboard, in quotes
dhh : d, h = gct.extract_gct(<'clipboard contents', like from above>)
(with the setup below -aimport ____ reimports those module before every function call, very helpful for those scripts that are actively evolving)
ipp : inplace=True) [modifying dataframes in place]
sepp : sep='\t') [formatting shortcut when saving data as txt]
lload : import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import sys, os, glob, gct, gcsv, pt
from pt import tolist
from pt import clip
from collections import Counter
%load_ext autoreload
%autoreload 1
%aimport plottools
%aimport plateanalysis
%aimport gt
%matplotlib

iPython

you can then view a function parameters / definition and the docstring (if defined) in ipython by putting ‘?’ instead of the function open parentheses

the ‘tolist’ function imported dircetly so it can be called without calling on the module first, takes a pasted column from Finder or Excel and returns a parsed list (if the names are plate-like, they’re truncated to short names - delimited at an underscore). remember to use triple quotes ‘’’ to capture multiple lines in ipython, then paste, close off with ‘’’) and your list is returned / captured by your variable.

```
l = tolist('''HOV-008
...: HOV-008
...: IC-975
...: IC-975
...: IC-975
...: IC-976
...: IC-976
...: IC-976''')
```

Summary Code

map summary

mapsummary.check_maps(path, compare=True, img=True, v=True, filt=True)

looks through .txt and .xlsx maps in a directory and summarizes their content and relationship with each other Generates plate visualizations of type and batch for each plate if img = True. V for verbose, lists names and doses per plate. if filter is true, just observe 6character name excel files, otherwise consider all files

csv summary

gcsv.summarize_csvs(path)

provide path containing csv files to generate output summarizing levels 1 and 10 for the plate as well as the posamp and ref

qc summary

qc.summarize_qc(path=’dflt’)

dflt path is desktop/newQC, but can be specified, will pulls together qc data for all qc folders in path, and outputs a concise summary as well as a comprehensive consolidation of all qc metrics for each plate/batch

Data Structures

dataframes

data packages have two components: the numerical data DataFrame, and the map / sample annotations as a DataFrame object, usually abbreviated to ‘d’ or ‘df’ and ‘h’ for data and header respecively.

the ‘dhh’ keyboard shortcut automatically extracts the gct and header file objects from a filepath currently within the clipboard. so after pressing F6 key (keyboard shortct to copy file path enclosed in single quotes to keyboard), in an iPython session typing ‘dhh’ will type out, and then paste and finish: d, h = gct.extractgct()

a preview header of a ZSVCQNORM file is shown below:

```
d.head():
‘FPA001:C03’ FPA001:C06 FPA001:C07 FPA001:C08 FPA001:C09
well
200814_at 0.79 –0.13 0.25 –0.61 –0.42
218597_s_at –0.09 0.88 0.91 0.91 –0.85
217140_s_at 0.78 0.17 –2.44 –0.74 0.21
209253_at 0.27 –1.28 0.64 –1.38 –1.76
214404_x_at 0.56 0.37 1.50 –0.86 –0.56
```

and the corrosponding header file information:

```
h.head()
batch cell plate well dose name type addr
FPA001:C03 A HepG2 FPA001 C03 0.0041 FP1413s test FP1:C03
FPA001:C06 B HepG2 FPA001 C06 0.33 FP1072c test FPA1:C06
```

FPA001:C07 A HepG2 FPA001 C07 0.11 FP0650c test FPA1:C07
FPA001:C08 B HepG2 FPA001 C08 0.037 FP1068c test FPA1:C08
FPA001:C09 A HepG2 FPA001 C09 1 FP0685s test FPA1:C09

So the index value of a header object is standardly defined as the well address, which is in the form <6char plate> : <3char well>. This info is also captured in the redundant ‘addr’ column in the map file, to provide for optional reindexing.

slicing and subsetting data

hsub

operations on the header object allow you to parse out wells by whatever criteria you’d like, and passing those samples as column selections for the dataframe allow you to extract any subset of the dataframe for further manipulation and plotting.

the standard form of using criteria for subsetting header dataframes and then passing the resulting index values into a dataframe works, but is cumbersome quite quickly:
hs = h[(h[‘type’]==’test‘) & (h[‘name’]==’foo’)]
ds = d[h.index.values]

so the command gt.hsub helps consolidate the header subset commands by passing in the header object and an argument dictionary
hs = gt.hsub(h, {’type‘:’test’, ‘name’:’foo’})

where the dictionary values are columns, and the values are the search terms. lists are also supported as dict values, in which case results that match any one of the values are included in the result
hs = gt.hsub(h, {’type‘:’test’, ‘name’:[‘foo1’, ‘foo2’] })

and the additional options of dsub, which takes in both the data and header dataframes and returns the subsets of both
ds, hs = gt.dsub(d, h, {’type‘:’test’, ‘name’:’foo’})

with a slight variant of gt.dosub (for data only) which only returns the subset data, better for straight inputs to plotting calls rather than continued dynamic work

```
ds = gt.dosub(d, h, {'type':'test', 'name':'foo'})
```

single plotting example
desc = {’well’: ‘E04’}
wells = gt.dosub(d, h, desc)
skyline.plot_skyline(wells)

breakdown

the other shortcut command which is helpful here is the gt.breakdown command, which helps to split up a dataframe into a series of chunks for plotting.

it will split out subsets of a passed data/header file combo broken out by any combination of batches, names, and doses. the categories are passed in as a string, with the attribute abbreviations ‘b’ for batch, ‘n’ for name, and ‘d’ for dose (‘w’ for well also supported). So ‘nd’ will partition by each unique name-dose combo, and return subsets of data and headers just correpsponding to those wells. Inclubing ‘bn’ for batch-name will treat each name in each batch separately.

```
results = gt.breakdown(d, h, ‘nd’) == > [ (ds1, hs1), (ds2, hs2)]
for ds, hs in results….
```

multiple plotting example w/ breakdown, will make two skyline plots, one of all cpd1 reps and the other of cpd2 reps:

```
desc = {’name’: [‘cpd1’, ‘cpd2’]}
wells = gt.dsub(d, h, desc)
for w in gt.breakdown(wells, ‘nd’)
skyline.plot_skyline(w)
```

forming consensus instances

the preferred consensus approach is a conservative one, which uses robust zscore against batch-matched vehicle wells to derive zscores. when then forming a consensus of multiple zscore instances, the expression value for each gene is taken to be the smallest magnitude as long as all measuerments are in the same direction (positive or negative), otherwise if there is any disagreement in direction the value is set to zero.

gt.consensus(df) will combine the sample instances in the passed dataframe and return the consolodated instance.

this can be paired with gt.breakdown() to assemble consensus datasets from sample collections with annotated replicates, and this pairing has been bundled into gt.assemble_consensus(df, h, cats):

```
dc, hc = gt.assemble_consensus(df, h, ‘nd’)
```

for datasets composed of multiple plates, those dataframe and header objects can be concatenated together and then passed into assemble_consensus. if plates don’t have annotations, but share layouts, then plates can be concatenated and then assemble consensus by ‘w’ which will match them up.

combining data

two or more data and header dfs in list formats can be combined as follows:

```
df = pd.concat(dflist, axis=1) // across columns
h = pd.concat(hlist, axis=0) // across rows
```

many data files can be conveniently loaded in consolidated fashion, but care must be taken such that the header types of those datafiles are the same, otherwise the combined header will be broken

```
d, h = gct.extract_multiple_gcts(pathlist)
```

the above currently has a shortcut built in to be able to paste a single string of POSIX paths copied from the Mac Finder and auto parse them apart to open and combine all gcts. Otherwise that should be a list of distinct file paths of gcts

Common Analyses

euclidean distance matrix

working with a data matrix, usually log2 data, and looking for global similarity picture. first sort the sample header information as desired:

```
h.sort_values([‘cell’,‘name’], ascending=[True,False],inplace=True
```

also of value is gt.hsub which allows you to flexibly subset a dataframe (usually a header file) by specifying a dict of arguments, as either strings or lists:

```
hA = gt.hsub(h, {’batch‘: [‘A’,‘D’], ‘name’:’vorinostat’})
```

then call (in this case with the data subset to hA as above)

```
plottools.plot_euclidean(df[hA], hA.index, fontsize=8, rot=, tick_denom=1)
```

passing in dataframe, the sample labels, and optionally specifying x-axis label rotation (default is 90, and spacing works best, though may be harder to read). fontsize may need to be adjusted to small (6~) for large numbers of samples. tick_denom specifies interval of labels, can be adjusted for n=x in groups of samples (still hard to do uneven #s of reps though).
capturing the resulting axes object from the above command allows you to modify that axis with commands, along with existing fig and plt commands.
plt.tight_layout() : auto arrange + re-size figure, usually better
ax.set_title/fig.title/plt.title(fontsize=) : set figure title, can set bigger fontsize

skylines

goal: visualize the differential expression across landmark genes for given condition. usually the zscore of an individual treatment, either alone or as a series of replicates which are each plotted along with the consensus profile

skyline.plot_skyline(wells, ighlights=None, outpath='dflt', t=1, maxv=10, title='auto', line1=None, line2=None)

highlights: list of genes to highlight on the resulting plot (v1 - highlights entire column slice with light grey, and plots resulting up/down in brighter color)
wells: argument you can pass in either one pd.Series object, or a list of Series, or a dataframe, and consensus will be produced if multiple profiles present.
t: threshold value for significance to plot features, default only plots values greater than +/- 1
maxv: y axis max, default is 10, can also be set to 'auto' which will auto fit the y axis to the local profile.
title, lines: support for file/image title, otherwise the name of wells[0], and two different lines below

plotting concentrations

plot_concentrations(f, genes='test2', mode='ind', label=False, outpath='dflt', incr=10)

f: can either be a gct file path, or a list object of [df, h] for the data of interest
genes: accepts a list of gene ids, otherwise can be set to *test1* for just one gene, *test2* for panel of 12 genes, or *test100* for a slice of 100 genes (all predefined)
mode: default is independent, to plot each replicate as independent point (multiple y values per x value), but can be 'avg' or 'med' to combine the reps into single y value
label: optional flag to print the name-batch-dose for each trt below the x-axis - joines name, dose, and batch
incr: a control for the x distance between points, but is a little funky with the auto scaling…

plottools.plot_concentrations([ds,hs], fn='this is a test', genes='test1',
label=True, incr=20, mode='med')

other plotting (histograms, pairplots)

handy commands available through seaborn include distplot for plotting a given distribution (of a single series of values):

sns.distplot(inst, hist=True, kde=False)

Where hist controlls the barplot and kde controlls a smoothed line representing the distribution. Multiple distributions can be plotted on the same axis, and the color will auto-increment.

pairplot is another handy command which will automatically plot all pairwise scatterplots of a passed in dataframe (note the axes are auto-scaled for each sub figure)

```
sns.pairplot(df)
```

deriving signatures & executing queries

signatures are usually derived using consensus profiles (see above).
to survey an instance and see number of genes passing thresholds, run

```
sigs.sig_survey(inst, cust=2.5) // where cust is a custom threshold
2:(38/15) 3:(23/4) 5:(13/0) 8:(6/0) 10:(3/0) 2.5:(31/6)
```

We can then see that 2.5 might indeed be an appropriate threshold to apply for a signature of this particular instance.

You can then either grab the signature into variable for the up and down list, or alternatively save the signature into grp files, optionally saving denominated

```
up_sig, dn_sig = sigs.get_sig(inst, 2.5)
or
sigs.save_sig(inst, 2.5, gs=False, name='dflt‘, path='dflt')
```

to then conduct a query using a signature on a given dataframe:

```
sigs.bulk_test_enrich(df, up, dn, h=None, outpath=False)
```

which will return a new dataframe in response, which includes the up score, down score, absolute enrichment and scaled enrichment within the local data.

passing header is optional, and will be included in the returned dataframe, if the outpath is defined the results will be saved instead of returned.