Week 10 - Lecture 19

István Albert
Huck Institutes for the Life Sciences
This week we explore a data analysis problem

- Consider the data in project GPL9270

- GSM are the series → data samples that correspond to similar conditions

- Task → find genes that are:
  - upregulated (in top 20%) in the majority of the samples in series 1
  - downregulated (in bottom 20%) in the majority of the samples in series 2
  - ignore missing data
Thought process

Up-regulated genes in all files of series 1

- all files → iterate with glob
- top, bottom genes → sort data
- common elements → intersect sets
- ignore data → filter
How to sort by multiple fields

Two ways about it

1. An easy way that is not initially obvious but it is very elegant and extremely fast.

2. A more complicated, traditional manner that involves creating a comparator function that returns +1, 0 and -1 depending on the result of a comparison.
Decorate → Sort → Un-decorate
Default tuple sort

```python
a = [(1, 'C'), (1, 'A'), (3, 'A'), (2, 'B')]
b = sorted(a)
print b
```

It is in the correct order by field 1 then by field 2!
Works for any number of fields

```python
a = [(1,2,1), (1,1,3), (2,3,1), (3,2,1),
    (3,1,2), (2,3,2)]

b = sorted(a)

print(b)
```

```
[(1, 1, 3), (1, 2, 1), (2, 3, 1), (2, 3, 2), (3, 1, 2), (3, 2, 1)]
```
DSU pattern

Decorate ➔ Sort ➔ Un-decorate

• It means that to sort an object by certain fields all we need is to create an extended (decorated) version of the data that contains fields that we want to sort by
Decorate-Sort example

```python
# sort this list by increasing name length

#

data = ['Jennifer', 'Janet', 'Jasmine', 'Jill']
```

Python output:
```
python -tu C:/cygwin/home/jalbert/sources/jalbert-web/ppt/week10/week19/demo.py 2>&1 returned 0
```
Decorate and sort

```python
def decorate(name):
    return len(name), name

names = [ 'Jennifer', 'Janet', 'Jasmine', 'Jill' ]
decor = map(decorate, names)
print decor

decor = sorted(decor)
print decor
```

```
[(8, 'Jennifer'), (5, 'Janet'), (7, 'Jasmine'), (4, 'Jill')]
[(4, 'Jill'), (5, 'Janet'), (7, 'Jasmine'), (8, 'Jennifer')]
```
```python
print decor

decor = sorted(decor)
print decor

def undecorate(elems):
    return elems[-1]

print map(undecorate, decor)
```

```
[(8, 'Jennifer'), (5, 'Janet'), (7, 'Jasmine'), (4, 'Jill')]
[(4, 'Jill'), (5, 'Janet'), (7, 'Jasmine'), (8, 'Jennifer')]
['Jill', 'Janet', 'Jasmine', 'Jennifer']
```
Onto our problem

```python
fname = './GPL9270/GSM455823.txt'
rows = bmbb.read_tabular(fname)
genes = bmbb.get_column(rows, 'ID_REF', mapfunc=str)
values = bmbb.get_column(rows, 'VALUE')

# decorated
decor = zip(values, genes)
decor = sorted(decor, reverse=True)
print decor[:10]
```
More meaningful aliases

```python
import bmmb

fname = '../GPL9270/GSM455823.txt'
rows = bmmb.read_tabular(fname)
genres = bmmb.string_column(rows, 'ID_REF')
values = bmmb.float_column(rows, 'VALUE')

# decorated
decor = zip(values, genres)
decor = sorted(decor, reverse=True)
print decor[:10]
```
Setting up aliases inside bmmb.py

```
88  89  def string_column(rows, colname):
90       return get_column(rows, colname, mapfunc=str)
91  
92  93  def float_column(rows, colname):
94       return get_column(rows, colname, mapfunc=myfloat)
95  
95  96  def last(elems):
97       # returns the last element
98       return elems[-1]
99  
```

Module also seem in the wild as ➔ PrepData, week4, week4version2, newweek4, week8
Find the top 10 highest values

```python
# decorate genes with values
decor = zip(values, genes)

# sort decorated list
decor = sorted(decor, reverse=True)

# undecorate sorted list of tuples
genes = map(bmmb.last, decor)

# top 10 genes by expression
print genes[:10]
```

Output:
```
['yddM', 'ydcM', 'ydcT', 'yddD', 'ydfS', 'ydjA', 'yebA', 'lrpB', 'yddG', 'rapI']
```
Homework 1/2

• Student data is stored as tuples representing name, gender and age:

  [ ('John', 'M', 22), ('Jack', 'M', 19), ('Jill', 'F', 21) ]

• Sort this data by gender, age then name to obtain:

  [( 'Jill', 'F', 21), ('Jack', 'M', 19), ('John', 'M', 22)]
1. For the GPL9270 data how many genes are present in the top quarter of expression (1000 top genes) in each of the experiments

• See possible output on next page
import bmmb, glob

# pick any file to get the starting set of genes
rows = bmmb.read_tabular('../GPL9270/GSM455823.txt')
genesis = bmmb.string_column(rows, 'ID_REF')
allgenes = set(genesis)

for fname in glob.glob('../GPL9270/GSM*.txt'):
    rows = bmmb.read_tabular(fname)
genesis = bmmb.string_column(rows, 'ID_REF')
values = bmmb.float_column(rows, 'VALUE')

# now intersect the top 1000 genes with the allgenes set
Power of simplicity
Our “toys”

Building blocks:

– List
– Tuples
– Sets
– Dictionaries

Glue them with: map, filter, zip, sorted, slice, min, max
Thought process

• It is not about algorithms
• Let the data structure do the work
• Know the type of different objects
• Make use of the existing structure
• Some of the upcoming slides will have “covered” parts

• Fill in the blanks so that the programs work
do not repeat yourself, repeat yourself

```python
import bmm

fname = '../GPL9270/GSM455823.txt'
rows = bmm.read_tabular(fname)
genres = bmm.string_column(rows, 'ID_REF')
values1 = bmm.float_column(rows, 'VALUE')

fname = '../GPL9270/GSM455824.txt'
rows = bmm.read_tabular(fname)
genres = bmm.string_column(rows, 'ID_REF')
values2 = bmm.float_column(rows, 'VALUE')

print values1[:2]
print values2[:2]
```

Command Output

```
[[-0.07115077899999997, -0.06737881099999997], [-0.234584873, -0.34200947999999998]]
```
import bmmb

def get_values(fname):
    rows = bmmb.read_tabular(fname)
    genes = bmmb.string_column(rows, 'ID_REF')
    values = bmmb.float_column(rows, 'VALUE')
    return genes, values1 = get_values('..\GPL9270\GSM455823.txt')
    genes, values2 = get_values('..\GPL9270\GSM455824.txt')

print values1[:2]
print values2[:2]
Find genes with the highest expression level change across two files

```python
genes, series1 = get_values('.../GPL9270/GSM455823.txt')
genes, series2 = get_values('.../GPL9270/GSM455824.txt')
collated = zip(genes, series1, series2)
def subtract(elems):
    name, val1, val2 = elems
    return val1 - val2, name
diffs = map(subtract, collated)
print diffs[:2]
```

`TypeError: unsupported operand type(s) for -: 'NoneType' and 'NoneType'`
Removing missing values

```python
def subtract(elems):
    name, val1, val2 = elems
    return val1 - val2, name

def remove(elems):

diffs = filter(remove, collated)
diffs = map(subtract, diffs)
diffs = sorted(diffs, reverse=True)
print diffs[:2]
```

```
[(0.6876380220000002, 'yebA'), (0.6761773039999998, 'ydfs')]
```
Still not quite right

- This only returns the relative difference, and that could go both ways, positive or negative

```python
def subtract(elems):
    name, val1, val2 = elems
    return val1 - val2, name
```

- We need to map a function that returns three elements, the **absolute difference**, **relative difference** and **gene name** → use `abs(value)`
Sorting by absolute differences

```python
def subtract(elems):
    name, val1, val2 = elems
    return absdif, reldif, name

diffs = filter(remove, collated)
diffs = map(subtract, diffs)
diffs = sorted(diffs, reverse=True)

print diffs[:2]
```

```
[(1.1810375440000001, -1.1810375440000001, 'yddC'),
 (1.0646327040000001, -1.0646327040000001, 'ydcR')]
```
Top 5 upregulated genes

```python
# continuing on the previous program
# we now have the sorted list of regulated genes

def positive(elems):
    #

def last(elems):
    #

updiffs = filter(positive, diffs)
upgenes = map(last, updiffs)

print updiffs[:2]
print upgenes[:5]
```

Output:
```
[(0.68763802200000002, 0.68763802200000002, 'yebA'),
 (0.67617730399999998, 0.67617730399999998, 'ydfs')]
['yebA', 'ydfs', 'yetG', 'yebD', 'veg']
```
How likely is it that the observed quantities are by chance

• If we know the expected error for the values in each dataset → we can directly estimate the likelihood of seeing a certain difference (statistical significance)

• If we do not know the errors a-priori, we have to estimate the errors ourselves – see next slides

• Next lecture will focus on statistical significance measures
Estimating errors

1. **Replicate experiments** → measure the same value multiple times

2. **Technical replicates** → same sample measured multiple times – usually easy

3. **Biological replicates** → multiple “identical” samples measured – more difficult

Typically we have a mixture of 2 and 3
Collecting the values for each gene

• Up to this point we treated each sample independently

• We now want to collect all the values for each gene, then operate on all values corresponding to a single gene

• We’ll start a new program for this
New starting point

```python
import bmmmb, glob

# get all sample names
allnames = glob.glob('..\GPL9270\GSM*.txt')

# moved the get_values() function into bmmmb.py
# run it on the first file name name
genes, values = bmmmb.get_values( allnames[0] )

# check that it actually works
print allnames[0]
print zip(genes, values)[:2]
```

```
../GPL9270\GSM455822.txt
[('opuCD', -0.202409751), ('ssuA', -0.248935244)]
```
We can map to dictionaries

```python
# moved the get_values() function into bmmb.py
# run it on the first file name name
genes, values = bmmb.get_values(allnames[0])

# recall the dictionary mapping
store = dict(zip(genes, values))

# first five keys
print store.keys()[:5]

# get one element
print store['opuCA']
```

Command Output

```
['yyCP', 'opuCB', 'opuCC', 'opuCA', 'opuCD']
-0.07215072
```
Dictionary containing lists as values

```python
a = {}
a['Jill'] = 123
print 'a =', a

b = {}
b['Jill'] = []
print 'b =', b

for num in range(5):
    b['Jill'].append(num)
print 'b =', b
```

```
a = {'Jill': 123}
b = {'Jill': []}
b = {'Jill': [0, 1, 2, 3, 4]}
```
Storage initialized, ready to be filled

```python
# get all sample names
allnames = glob.glob('./GPL9270/GSM*.txt')

genes, values = bmbb.get_values( allnames[0] )

# initialize our storage to contain empty lists
store = {}
for gene in genes:
    store[gene] = []

print store.keys()[:5]
print store['opuCA']
```
Same value – different containment

```python
store = {}
for gene in genes:
    store[gene] = []

# fill in the values
collate = zip(genes, values)
for gene, value in collate:
    store[gene].append(value)

# it is the same value as two slides back, but in a list
print store.keys()[:5]
print store['opuCA']
```

Command Output

```
['yyCP', 'opuCB', 'opuCC', 'opuCA', 'opuCD']
[-0.07215072000000002]
```
store = {}
for gene in genes:
    store[gene] = []

# let's do if for the first three only
for name in allnames[:3]:
    print 'Processing file %s' % name
    genes, values = bmm.get_values( name )

print store.keys()[:5]
print store['opuCA']

['yyCP', 'opuCB', 'opuCC', 'opuCA', 'opuCD']
[-0.07215072000000002, -0.048386948999999999, -0.022444268]