Lab 2

Tabular data processing with Pandas

In this lab exercise we will look at how to work with data stored in a tabular form and perform exploratory data analysis on it. We will be using the Python Data Analysis Library or Pandas to do this.

At the end of the exercise, remember to fill out the response form here.

Introducing Pandas

The two main data structures that Pandas supports are Series and DataFrames. Series are one-dimensional data structures that are a collection of any data type. DataFrames on the other hand are two dimensional data structures which resemble a database table or say an Excel spreadsheet. In this lab we will primarily be using DataFrames and will look at operations that we can perform using them.

Before you start, download the following file to your home directory (i.e `/home/datascience`) and untar it.

```
wget --no-check-certificate https://raw.github.com/biddata/datascience/master/F14/lab2/data/wc_day6_1_sample.tar.bz2
tar -xf wc_day6_1_sample.tar.bz2
```

After that run `ipython notebook` from your command line. This will launch a browser window with the `iPython Dashboard`. IPython notebook provides an interactive environment for using Python. Click on `New Notebook` to create a new notebook. We'll first import `pylab`, a library that supports plotting and direct plots to be shown line. Next we will import the Pandas library and create a simple DataFrame, by running

```
from pylab import *
%matplotlib inline
import pandas as pd
df = pd.DataFrame( { 'a' : [1, 2, 3, 4], 'b': [ 'w', 'x', 'y', 'z'] })
```

NOTE: In the iPython notebook you'll need to use `Cntrl + Enter` or click the play button to execute code in a cell.

If you need clarifications about the Pandas API you can type the function name followed by `?` to get inline help. For example to get help with the above call run:

```
pd.DataFrame?
```

If you want to see the same in a browser lookup the function in the API documentation.

DataFrame basics

The simplest way to see what is in a DataFrame is to just print it to the console. For example to see the DataFrame we created before you can just type `df` and see something like

```
df
   a  b
0  1  w
1  2  x
2  3  y
3  4  z

[4 rows x 2 columns]
```

This shows that we have two columnns 'a' and 'b' and four rows in our DataFrame.

However large DataFrames cannot be printed to the console and we have higher level commands to inspect its contents. To get information on the schema of the DataFrames, we can use the `info` function

```
df.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 4 entries, 0 to 3
Data columns (total 2 columns):
a    4  non-null values
b    4  non-null values
dtypes: int64(1), object(1)
```

To see the first few rows you can use `head` and to see the last few rows you can use `tail`. This is similar to the UNIX-command line tools (Remember Lab 1 !?)

```
In [31]: df.head(2)
Out[31]:
   a  b
0  1  w
1  2  x

[2 rows x 2 columns]
```

To print any range of rows from the DataFrame you can use array-like indexing of row ids. As you might have noticed rows are numbered from 0 in Pandas, so to get the middle two rows we can use the range `1:3`

```
In [34]: df[1:3]
Out[34]:
   a  b
1  2  x
2  3  y

[2 rows x 2 columns]
```

Finally, Pandas also has a useful function `describe` that summarizes the contents of numerical columns in a DataFrame. For example in `df` we can see the mean, standard deviation etc. for the column `a` by running `describe`.

```
In [42]: df.describe()
Out[42]:
              a
count  4.000000
mean   2.500000
std    1.290994
min    1.000000
25%    1.750000
50%    2.500000
75%    3.250000
max    4.000000

[8 rows x 1 columns]
```

Having worked our way through the basics, lets now see how we can use Pandas for data analysis. To do this part of the lab we will reuse the World Cup soccer logs from Lab 1. However this time the input data has been sampled and formatted as a `csv` file that you will load first.

```
log_df = pd.read_csv("/home/datascience/wc_day6_1_sample.csv",
                     names=['ClientID', 'Date', 'Time', 'URL', 'ResponseCode', 'Size'],
                     na_values=['-'])
```

The `names` argument tells Pandas what the column names are in our file and `na_values` indicates what character is used for missing values in our dataset. Use the commands from the previous section to explore the dataset and its summary statistics.

DIY

1. How many rows are present in `log_df` ?
2. What are the URLs between rows 85 and 90 ?

SQL-like operators

Next we will look at operators in Pandas that allow us to perform SQL-like queries on the dataset.

Selection

A SQL statement typically selects a subset of rows from a table that match a given criteria. This is known as the Selection operator in Relational Algebra. Similarly we can perform selections in Pandas using boolean indexing.

Boolean indexing refers to a technique where you can use a list of boolean values to filter a DataFrame. For example lets say we only want entries from '01/May/1998'. To do this we can create a boolean list like

```
is_may1st = log_df['Date'] == '01/May/1998'
is_may1st.head(2)
0    False
1    False
Name: Date, dtype: bool
```

Now we can filter our DataFrame by passing it the boolean list.

```
may1_df = log_df[is_may1st]
may1_df.head()
```

Or we can directly do this by passing in the boolean clause to the DataFrame

```
may1_df = log_df[log_df['Date'] == '01/May/1998']
may1_df.head()
```

Projection

While selection is used for filtering rows, projection is the relational algebra operator used to select columns. To do this with Pandas we just need to pass in a list of columns that we wish to select. For example to only keep the 'URL' and 'ResponseCode' column we would run

```
url_codes = log_df[['URL', 'ResponseCode']]
url_codes.head(5)
```

Grouping

Pandas also allows you to group the DataFrame by values in any column. For example to group requests by 'ResponseCode' you can run

```
grouped = log_df.groupby('ResponseCode')
grouped
<pandas.core.groupby.DataFrameGroupBy object at 0xbda56ec>
```

As you can see from the output above, `grouped` is not a `DataFrame` but an object of type `DataFrameGroupBy`. This just means that it contains a number of groups and each group is in turn a DataFrame. To see this try

```
grouped.ngroups
grouped.groups.keys()
  [400.0, 200.0, 302.0, 304.0, 404.0, 206.0, 500.0]
grouped.get_group(200).head()
   ClientID         Date      Time                                    URL ResponseCode
0      1044  30/Apr/1998  22:46:12                     /images/11104.gif          200
1     10871  01/May/1998  12:10:53                     /images/ligne.gif          200
2     11012  01/May/1998  12:17:30  /english/individuals/player111503.htm          200
5     13649  01/May/1998  14:55:01                /images/hm_anime_e.gif          200
6     15006  01/May/1998  16:14:32  /english/images/comp_bu_group_off.gif          200
```

You can also group by multiple columns by just passing the a list of column names. For example to group by both date and response code you can run

```
multi_grouped = log_df.groupby(['ResponseCode', 'Date'])
```

Pandas also has useful commands to print various statistics about elements in each group.

1. `grouped.describe()` prints summary statistics for numeric columns in each group
2. `grouped.size()` prints the number of elements in each group
3. Similarly `grouped.sum()`, `grouped.mean()` and `grouped.median()` print the sum, mean and median values for numeric columns in each group

Other SQL operators

We have only looked at the basic SQL operators so far. Pandas also supports operations like `sort`, `join` and `indexing` to support a wide-range of queries. You can read more about this and try out examples with the Pandas comparison to SQL

DIY

1. Lab 1 Deja vu ! Use selection to print the number of requests that had HTTP return code 404.
2. How many of them were on 30th April and how many on 1st May ? Hint: Use the `multi_grouped` DataFrame we created above.

Applying functions to rows, column

So far we have been using SQL-style operators to process our data. However to do data cleaning or more complex analysis we often need to apply functions on row or column of a DataFrame.

For example, consider the columns 'Date' and 'Time' in our Dataframe. uld be useful if we could combine these columns and create `datetimects then it would be useful for filtering, grouping etc.

To create a `DateTime` column we will use Pandas helper function `to_datetime`. This function takes a string and converts it to a `datetime` object. To call this on every row of the DataFrame, we use the `apply` function. `apply` takes two arguments, the first a function to apply and secondly `axis` which indicates if this should be applied on every row (axis=1) or column (axis=0)

```
log_df['DateTime'] = pd.to_datetime(log_df.apply(lambda row: row['Date'] + ' ' + row['Time'], axis=1))
```

This might take a minute to run as we are adding a new column for every request in our table. Meanwhile take a look at the various components of `DateTime` objects in the API documentation. For example if we want to group by hour we can now use the DateTime API instead of doing any string parsing.

```
hour_grouped = log_df.groupby(lambda row: log_df['DateTime'][row].hour)
hour_grouped.ngroups
```

Finally, note that you can apply operations on each group using the `apply` method. This is similar to the `apply` on the DataFrame we saw earlier except the `apply` method is now called *once per group*.

DIY

1. Create a new column that contains the ResponseSize in kilo bytes.

Plotting data in a DataFrame

In a future lab exercise we will look at plotting in greater detail. However we can produce simple plots using the Python `matplotlib` library with a DataFrame. For example to plot a Series or a DataFrame you can just call `plot()` on the object and for a histogram just call `hist()`

```
rand_df = pd.DataFrame({'a' : randn(100)})
rand_df.plot()
rand_df.hist()
```

Exercises

1. What is the average file size for images (.gif or .jpg or .jpeg files) which had response code 200 ? What is the standard deviation ?
2. Generate a histogram of traffic to the site every *half-hour* and plot this.

Challenge exercises (Optional)

1. We wish to see if there is any correlation between client-ids and hours of the day at which they visit the website. Get 100 random client ids from the dataset and plot a scatter plot that shows the hours of the day these clients sent requests.
2. The log file used in the lab was from one day of the WorldCup. Lets apply our analysis to another day's logs. For this start with the raw log file `https://raw.github.com/biddata/datascience/master/F14/lab2/data/wc_day91_1_log.tar.bz2` from github and load it as a Pandas DataFrame. Repeat the first 2 exercises with it. How similar or different are the results ? Hint: You can use UNIX command line tools from Lab 1 to first get a csv file and then load it into Pandas.

Lab 2 Responses

Remember to fill out your responses here.