R/RStudio/RCpp Tips and Tricks:

1. **Auto Completion**: Begin typing an R command and then hit the tab key: R Studio will fill out the rest of the command for you.
2. **Close parenthesis and quotation highlighting:** RStudio takes care of unclosed parenthesis.
3. **Disable automatic insertion of closing parenthesis/quotation mark**: By default, R Studio will automatically insert a closing parenthesis or quotation mark when you begin typing an opening parenthesis or quotation mark. This can be turned off from the 'Tools'.
4. **Enable code wrapping**: Any part of your R script that would usually fall off the side of your screen is, instead, continued on the following line.
5. **Installing packages from menu:** Packages can be installed from the side menu.
6. **Enabling packages from side-menu:** A convenient way to load up R packages in R Studio is to simply check/uncheck the relevant boxes in the 'Packages' side-menu. This menu also allows you to quickly take inventory of all your active packages, which may be useful in situations where you need to free up some memory or debug a package incompatibility.
7. **Set working directory from side-menu:** Instead of typing the setwd() command you can set your working directly via the 'Files' side-menu by first navigating to your desired file location and then simply selecting the 'Set As Working Directory' option from the 'More' drop-down menu.
8. **Copy+paste from plot view:** It is possible to copy and paste a plot image directly from the zoom'd plot view without writing it to a .pdf or .png file first. Simply "Zoom" an active plot from the side-menu and right-click the plot view and select the "Copy Image" command.
9. **Quick Commenting:** Last, but certainly not least, you can comment out your code in R Studio by highlighting the offensive lines and simply hitting ctrl+shift+c. To un-comment, simply re-highlight and hit ctrl+shift+c again.

Creating a package from scratch in R:

1. Install "devtools" and "roxygen2".
2. Make sure your default directory is empty.
3. Use *setwd()* to set your current working dir where you want to create the package.
4. Use *create("myPackage")* to create an empty package called "mypackage" in your desired location.
5. Edit the .R file that is created.
6. Add documentation using the roxygen2 documentation package.
7. Process your documentation to using the *document()* command.
8. Install your package: Can be done using command or the GUI on the right side.
9. To use GUI, build the package using, *R CMD build mynewpackage* and then create the .tar file using the command *R CMD INSTALL mynewpackage_0.1.tar.gz*
10. Check Documentation.
11. Enjoy!

Python Tips and tricks: “Use PyCharm IDE”

1. In place swapping of two numbers:

```
x, y = 10, 20
print(x, y)

x, y = y, x
print(x, y)

#1 (10, 20)
#2 (20, 10)
```

2. Chaining of comparison operators:

```
n = 10
result = 1 < n < 20
print(result)

# True

result = 1 > n <= 9
print(result)

# False
```

3. Use of ternary operator

```
[on_true] if [expression] else [on_false]
```

   a. for conditional assignment:

```
def small(a, b, c):
        return a if a <= b and a <= c else (b if b <= a and b <= c else c)

print(small(1, 0, 1))
print(small(1, 2, 2))
print(small(2, 2, 3))
print(small(5, 4, 3))

#Output
#0 #1 #2 #3
```

   b. for list comprehension:

```
[m**2 if m > 10 else m**4 for m in range(50)]

#=> [0, 1, 16, 81, 256, 625, 1296, 2401, 4096, 6561, 10000, 121, 144, 169, 196, 225, 256, 289,
324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784, 841, 900, 961, 1024, 1089, 1156, 1225,
1296, 1369, 1444, 1521, 1600, 1681, 1764, 1849, 1936, 2025, 2116, 2209, 2304, 2401]
```

4. Dictionary/Set Comprehension:

```
testDict = {i: i * i for i in xrange(10)}
testSet = {i * 2 for i in xrange(10)}

print(testSet)
print(testDict)

#set([0, 2, 4, 6, 8, 10, 12, 14, 16, 18])
#{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

5. Simplify If statement

```
if m in [1,3,5,7]:
```

instead of:

```
if m==1 or m==3 or m==5 or m==7:
```

6. Using enumeration method: To find an index while you're inside a loop.

```
testlist = [10, 20, 30]
for i, value in enumerate(testlist):
        print(i, ': ', value)

#1-> 0 : 10
#2-> 1 : 20
#3-> 2 : 30
```

7. Return multiple values from functions

```
# function returning multiple values.
def x():
        return 1, 2, 3, 4

# Calling the above function.
a, b, c, d = x()

print(a, b, c, d)

#-> 1 2 3 4
```

8. Create dictionary from two related sequences –

```
t1 = (1, 2, 3)
t2 = (10, 20, 30)

print(dict (zip(t1,t2)))

#-> {1: 10, 2: 20, 3: 30}
```

9. Modules

```
# 1- Module definition => save file as my_function.py
def minmax(a,b):
    if a <= b:
        min, max = a, b
    else:
        min, max = b, a
    return min, max
```

```
# 2- Module Usage
import my_function
x,y = my_function.minmax(25, 6.3)

print(x)
print(y)
```

10. Zipping and unzipping lists and iterables

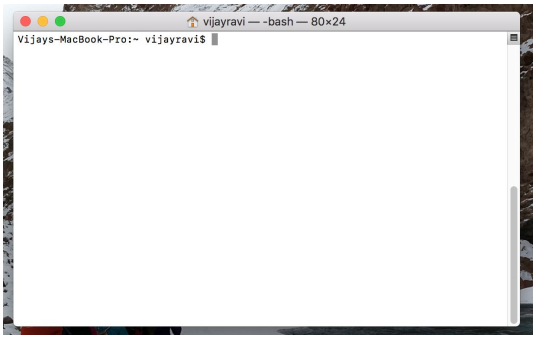
```
>>> a = [1, 2, 3]
>>> b = ['a', 'b', 'c']
>>> z = zip(a, b)
>>> z
[(1, 'a'), (2, 'b'), (3, 'c')]
>>> zip(*z)
[(1, 2, 3), ('a', 'b', 'c')]
```

STATS 202A : STATISTICS PROGRAMMING

HOMEWORK 8: UNIX COMMANDS

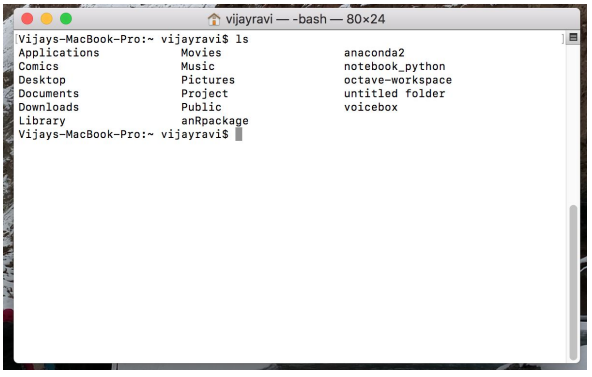
NAME: **VIJAY RAVI** UID: **805033666**

Command Terminal on MAC where UNIX commands are executed.

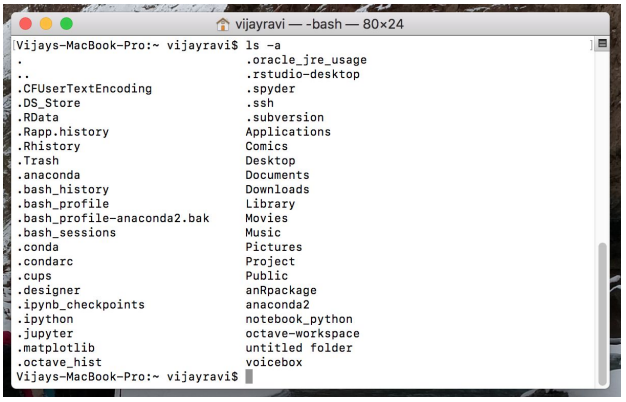


The following UNIX commands have been tried. Attached is the screenshot with each and brief explanation of each command functionality.

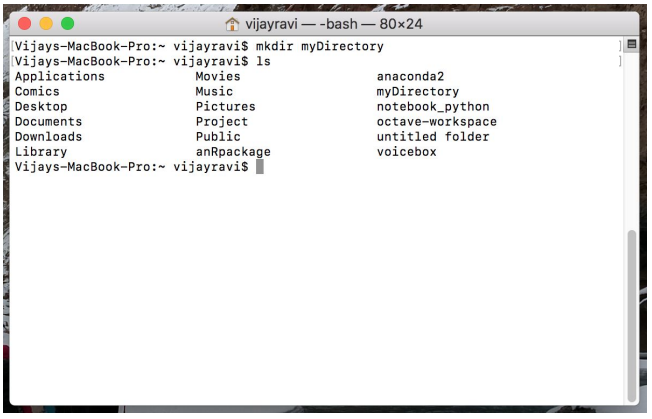
1. '**ls**': The **ls** command (lowercase L and lowercase S) lists the contents of your current working directory.



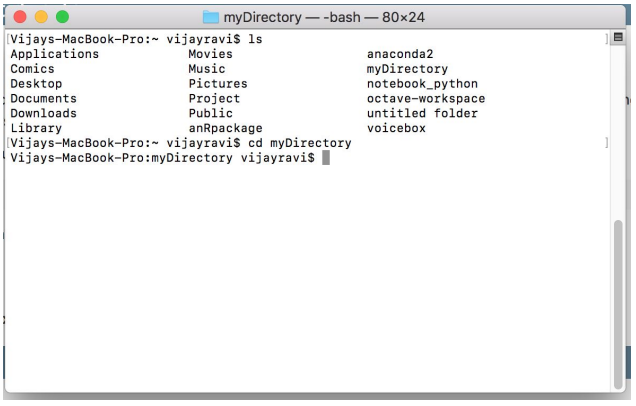
2. '**ls -a**' : List all files including the hidden files.



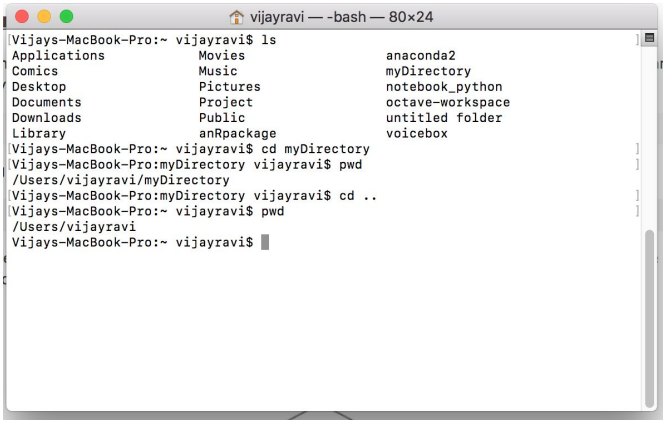
3. **‘mkdir**: make a directory. Creating a directory called ‘myDirectory’. Using ‘ls’ to check if directory has been created.



4. **‘cd *directory*’** : change working directory to the named ***directory***. Changed the working directory to the newly created directory.

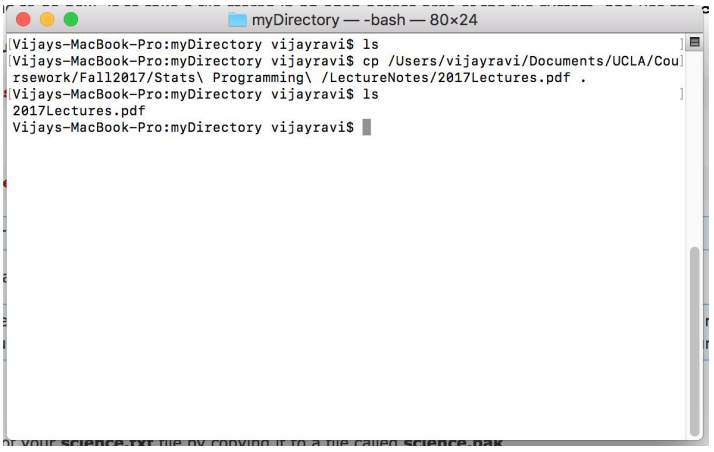


5. **‘cd’** : Change working directory to home directory.
6. **‘cd ..’** : Change working directory to parent directory.
7. **‘pwd’** : Display the path of the current working directory.

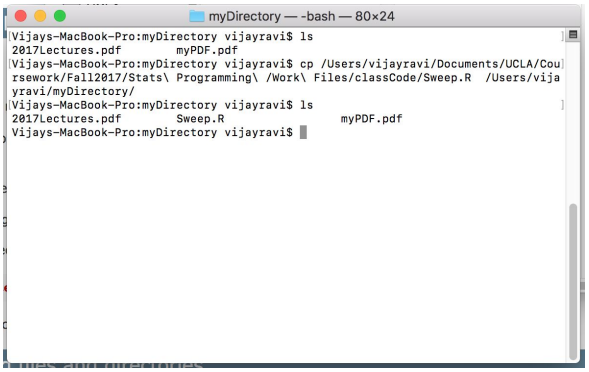


Changed the working directory back to parent directory. Confirmed the change by printing the path.

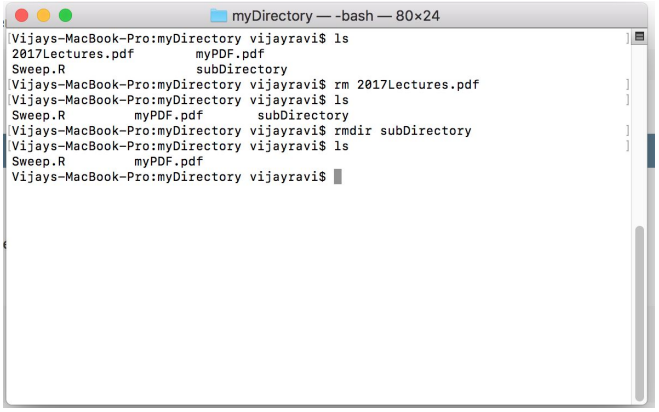
8. ‘Cp file1 file2’: Copy file1 and rename it as file2. Here I am copying the lecture notes from its file location to the current working directory. Using ‘ls’ to verify if file exists in copied location. Using the (.) notation for current working directory. ***myDirectory*** is empty before copying but has the ***2017Lectures.pdf*** after file has been copied.



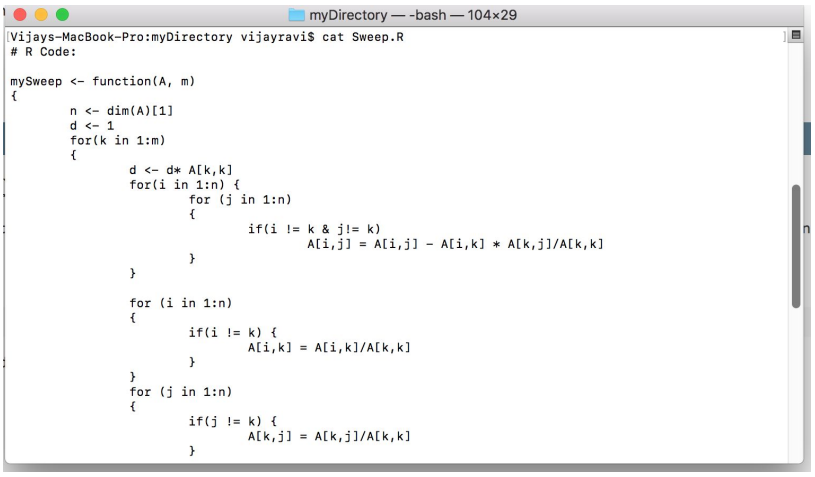
9. ‘mv file1 file2’: moves (or renames) **file1** to **file2.** Here, I am moving Sweep.R from it’s location to ‘myDirectory’. If the file is in the same directory, mv command can be used to rename the file.



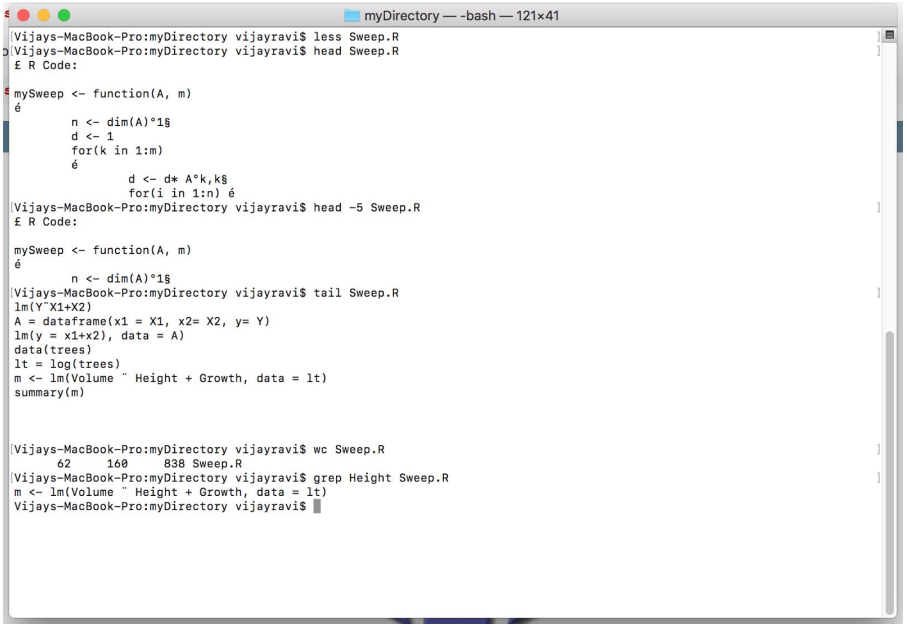
10. ‘rm ‘ and ‘rmdir’: These commands are used to remove/delete a file or a directory. Here I am removing the file ***2017Lectures.pdf*** and the directory ***‘subDirectory’.***



11. ‘clear’: clears the command prompt/terminal. History is still accessible by up/down cursor key.
12. ‘cat’: The command cat can be used to display the contents of a file on the screen. Here, I am displaying the contents of the sweep.R file.

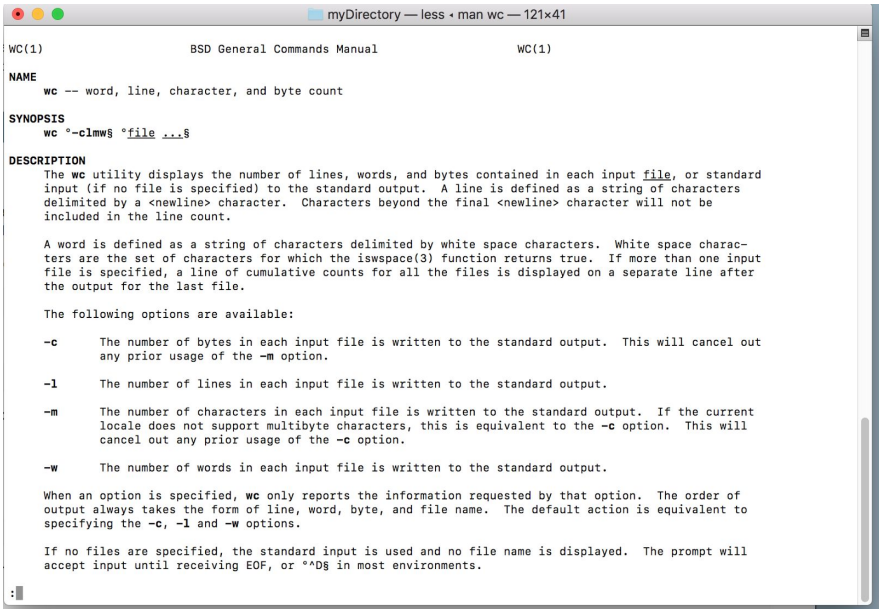


13. ‘less file1’ : Displays the contents of file1 one page at a time. Press spacebar to navigate to the next page. Press q to exit reading the file.
14. ‘head file1’: Displays first 10 lines of the file.
15. ‘tail file1’: Displays last 10 lines of the file.
16. ‘grep ***keyword*** file1’ : Returns the occurrence of the keyword in the file.
17. ‘wc file1’: Returns the count number of lines/words/characters in file

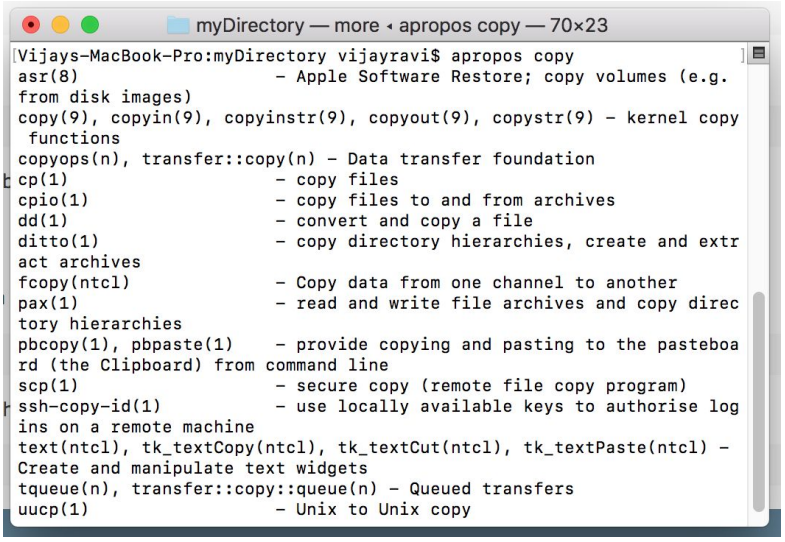


The file Sweep.R has the word ‘Height’ and has 62 lines, 160 words and 838 characters.

18. ‘man *command*’ : read the online manual/Documentation page for a command. Use cursor to navigate. Press ‘q’ to exit. ‘**man wc’** gives the following output:

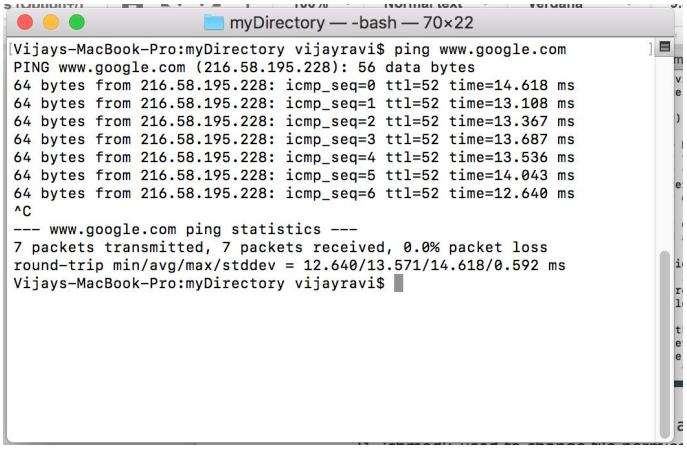


19. ‘apropos *keyword*’ : match commands with keyword in their man(manual) pages. ‘apropos copy’ gives the following output:

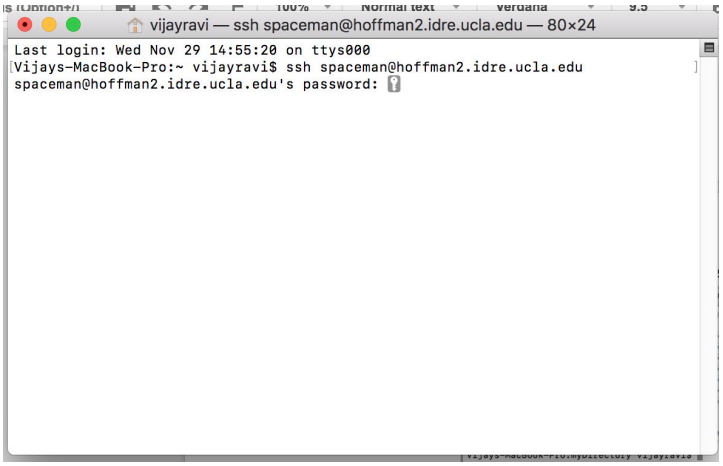


20. ‘vim file1’: Vim can be used to edit a file. Vim is a text editor.
21. ‘chmod’: used to change file permissions (read/write/executable).
    Eg: chmod 777 – read, write, execute for all

22. ‘ping host’ – ping host and output results. Press ‘CTRL+c’ to exit ping.



23. ‘ssh user@host’ – connect to host as user.



24. Ctrl+C – halts the current command
25. Ctrl+Z – stops the current command, resume with fg in the foreground or bg in the background
26. Ctrl+D – logout of current session, similar to exit
27. Ctrl+W – erases one word in the current line
28. Ctrl+U – erases the whole line
29. Ctrl+R – type to bring up a recent command !! - repeats the last command
30. exit – log out of current session

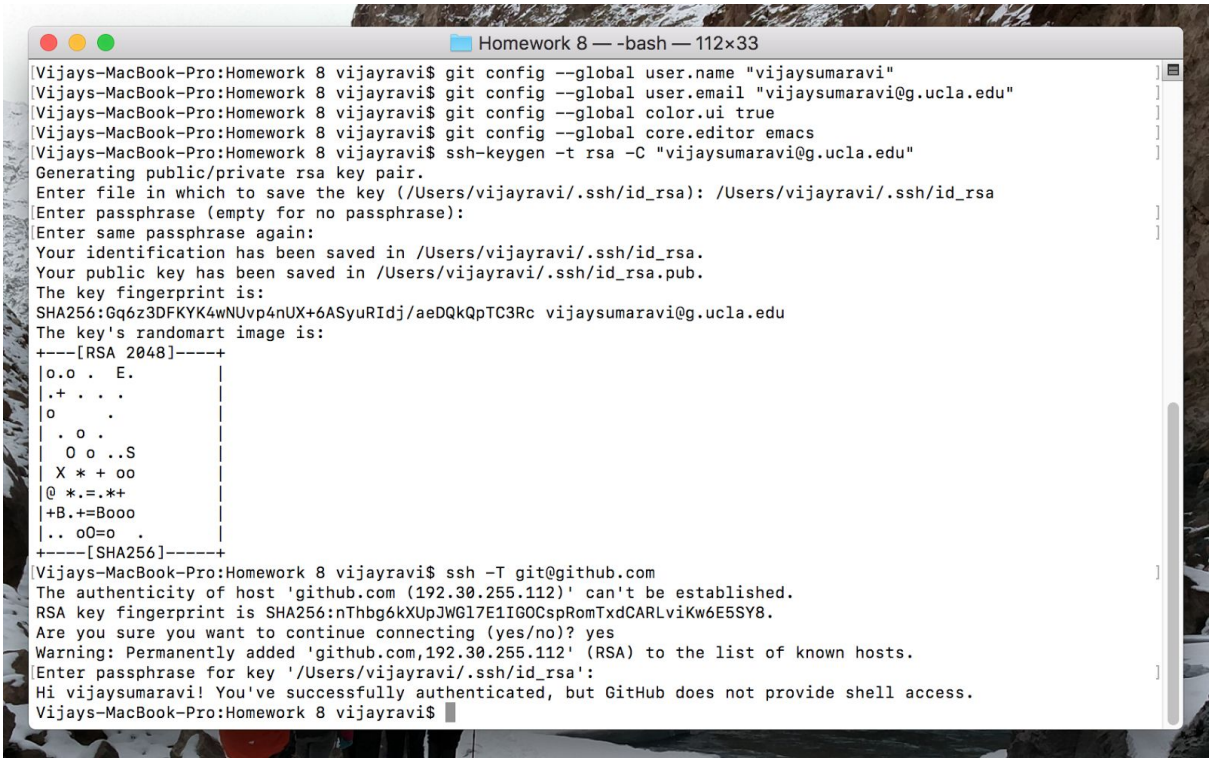
STATS 202A: STATISTICS PROGRAMMING

HW8: GITHUB COMMANDS

NAME: **VIJAY RAVI** UID: **805033666**

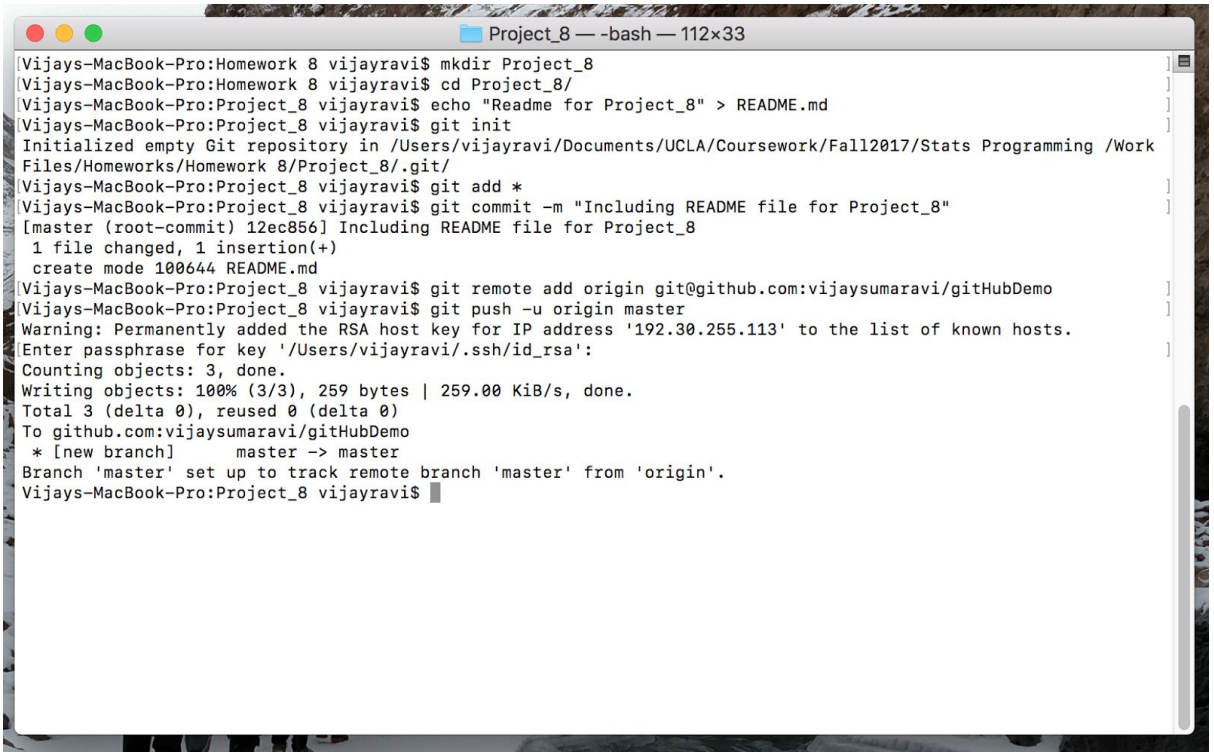
STEP 1: Create an account on https://github.com/. Account created: vijaysumaravi

STEP 2: Use the following commands as in the screenshot to set up GITHUB on your system.



STEP 3: Creating a new Repository from scratch

- Create a new repository in https://github.com/
- I have named the repo as “gitHubDemo”
- Follow the following steps in the screenshot to setup the new project, “Project_8” to the repository.

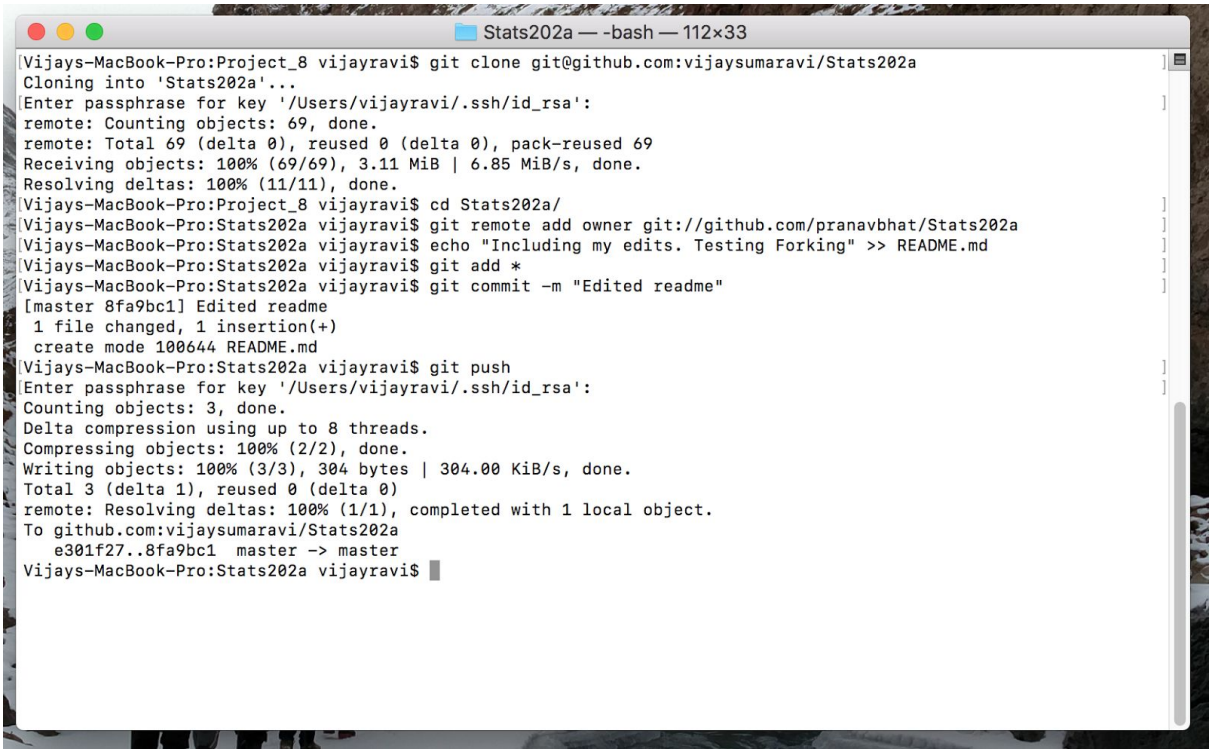


STEP 4: Continuing development and keeping the repository in sync with local development environment.

- Once a piece of developed code is tested and stable follow the below mentioned steps to push the changes to the repository.
- git add filename.extension
- git commit -m “Pushing new feature X”
- git push

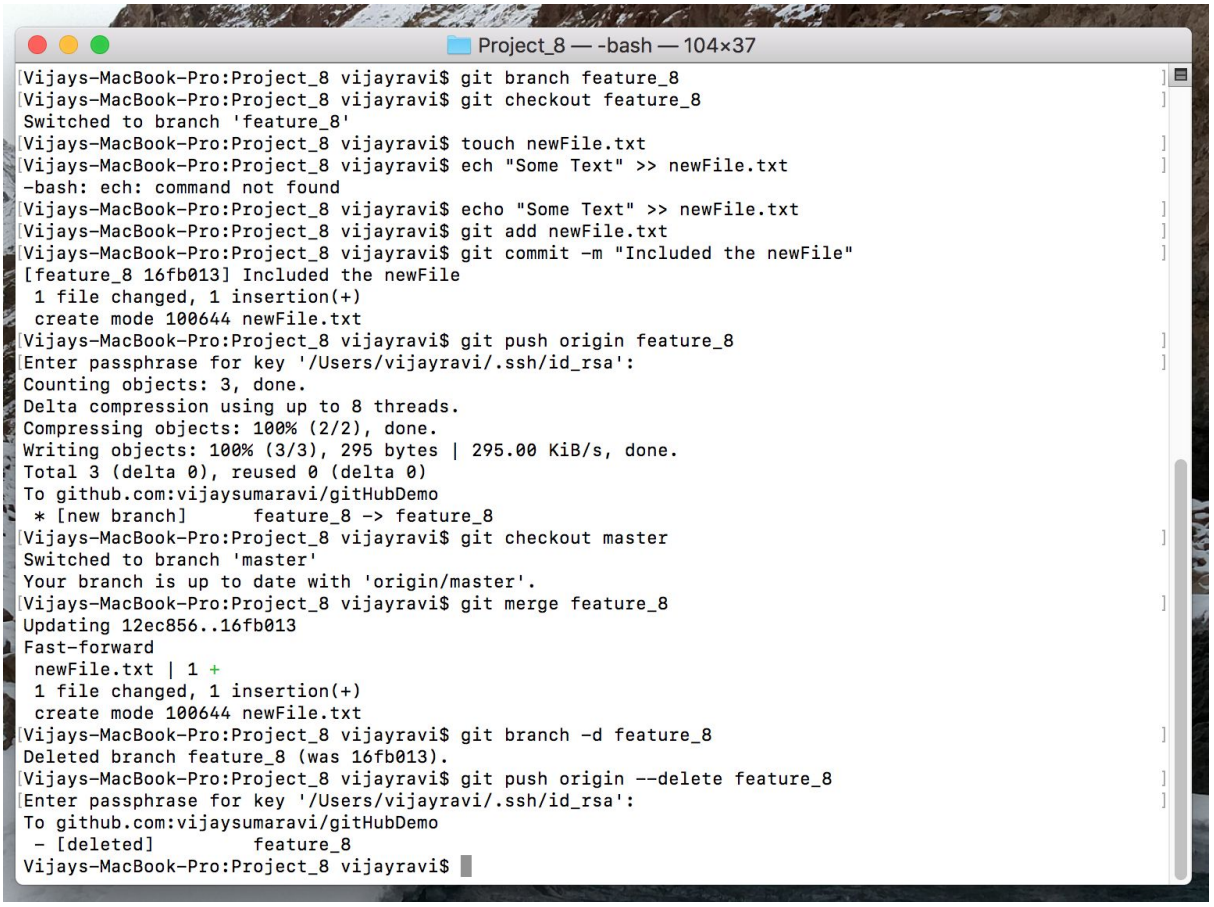
STEP 5: Forking someone else's repository.

- Fork the repository on https://github.com
- Clone the repository to the local system and continue development, push changes.
- I have forked "Stats202a' from pranavbhat and changed the README.md file.



STEP 6: Branching

- Creating a new feature branch.
- Developing and Testing in the branch.
- Merging development in the new branch on to the master branch.
- Deleting the new feature



SAS® Cheat Sheet

SAS Language

ATTRIB *var$_n$* <LENGTH='*var$_n$-length*'> <LABEL='*var$_n$-label*'> <FORMAT=*var$_n$-format.*> <INFORMAT=*var$_n$-informat.*>;
Associates a format, informat, label, and/or length with one or more variables

CARDS or **CARDS4** | **DATALINES** or **DATALINES4**
Indicates that data lines follow (suffix of 4 if data has ';'s).

DATA <*dset$_n$* <(*dset-options$_n$*)>>;
Begins a DATA step and provides names for any output SAS data sets. See Data Set Options for options that are available in the DATA statement.

DO *index-var=start_value* **TO** *end_value* <**BY** *step*>;

DO UNTIL (*expression*);

DO WHILE (*expression*);
Groups a set of statements as a single unit. Note that UNTIL conditions are evaluated at the end of the loop and thus execute at least once.

FILE *filename* <*options*>**;**
Specifies the current output file for PUT statements. Options include:

MOD	output is appended to an existing file.
OLD	output overwrites an existing file.

IF *expression* **THEN** *statement*; .... <**ELSE**> *statement*;
SAS evaluates the expression in an IF statement to produce a result that is either non-zero, zero, or missing. If result >0 then TRUE, else FALSE.

INFILE *filename* <*options*>**;**
Specifies an external file to read with an INPUT statement. Options include:
DELIMITER|DLM= *delimiters*
Specifies a delimiter for list input.
LENGTH= variable
Names a variable that SAS sets to the length of the current input line.

INPUT *var*<=> <*$*> *startcol* <*-endcol*> <*.dec*> <@ | @@>;

INPUT <*pointer-control*> *variable informat.* <@ | @@>*;*

INPUT <*pointer-control*> *variable* <*$*> <*&*> <@ | @@>*;*
Input records from the current input file, placing the values into SAS variables.

MERGE *ds1* <(*doptions*)> <*... dsn*<(*doptions*)>> <END=*var*>;
Joins observations from two or more SAS data sets into single observations.

OUTPUT<*data-set-name(s)*>;
Writes the current observation to a SAS data set.

PUT *var*<=> <*$*> *startcol* <*-endcol*> <*.dec*> <@ | @@>;

PUT <*pointer-control*> <"*text*"|*variable format.*> <@ | @@>;
Writes variable values and/or text to the output line.

RETAIN *variable$_n$* <*initial-value$_n$*>;
Causes a variable to retain its value from one iteration of the data step to the next.

SET <*data-set(s)* <(*data-set-options(s)*)>> <*POINT=varname*> <*NOBS=varname*> <*END=varname*>*;*
Reads observations from one or more data sets.

Sum: *variable+expression*
Adds the result of an expression to an accumulator var.

TITLE *<n> <"text">;*
Specifies title lines for SAS output. n specifies the relative line number with n being between 1 and 10.

WHERE *where-expression;*
Selects observations from SAS data sets that meet a particular condition that is true.

SAS Data Set Options

DROP=*variable(s)* Excludes variables from processing.

FIRSTOBS=*n* Specifies the first observation to process

IN=*variable* Creates and names a variable that indicates whether the data set contributed data to the current observation.

KEEP=*variable(s)* Selects variables for processing.

LABEL='*label*' Specifies a label for a SAS data set

OBS=*n* Specifies the first n observations to process

POINT=*variable* Direct observation number variable

RENAME=(*oldname$_1$=newname$_1$* <*...oldname$_n$=newname$_n$*>)
Changes the name of a variable.

WHERE=(*expression$_1$* <*logical-operator expression$_n$*>)
Selects observations from a SAS data set that meet certain conditions before SAS brings them into the DATA or PROC step for processing.

SAS Functions

BYTE(*n*) Returns one character in the ASCII or EBCDIC collating sequence where n is an integer representing a specific ASCII or EBCDIC character

COMPBL(*source*) Removes multiple blanks from a character string

COMPRESS(*source*<*,characters-to-remove*>)
Removes specific characters from a character string

DATE() Returns the current date as a SAS date value

DATEPART(*datetime*) Extracts the date from a SAS datetime value

DATETIME() Returns the current date and time of day

DAY(*date*) Returns the day of the month from a SAS date value

HMS(*hour,minute,second*) Returns a SAS time value from hour, minute, and second

INDEX(*source,excerpt*) Searches the source for the character string specified by the excerpt

LEFT(*argument*) Left-aligns a SAS character string

LENGTH(*argument*) Returns the length of an argument

LOWCASE(*argument*) Converts all letters in an argument to lowercase

MAX(*argument,argument, ...*) Returns the largest value of the numeric arguments

MDY(*month,day,year*) Returns a SAS date value from month, day, and year

MIN(*argument,argument, ...*) Returns the smallest value of the numeric arguments

MISSING(*argument*) Indicates whether the argument contains a missing value

MOD(*argument$_1$, argument$_2$*) Returns the remainder

MONTH(*date*) Returns the month from a SAS date value

RANK(*x*) Returns the position of a character x in the ASCII or EBCDIC collating sequence

REPEAT(*'character-expression',n*) Repeats a character expression n+1 times.

RIGHT(*argument*) Right-aligns a character expression

ROUND(*argument,round-off-unit*) Rounds to the nearest round-off unit

SCAN(*argument,n*<*,delimiters*>) Returns a given word from a character expression

SUBSTR(*argument,position*<*,n*>) Extracts a substring from an argument.

TIME() Returns the current time of day

TIMEPART(*datetime*) Extracts a time value from a SAS datetime value

TODAY() Returns the current date as a SAS date value

TRANSLATE(*source,to,from*) Replaces specific characters in a character expression

SUM(*argument,argument, ...*) Returns the total value of the numeric arguments

TRIM(*argument*) Takes the argument and removes any trailing blanks.

UPCASE(*argument*) Converts all letters in an argument to uppercase

WEEKDAY(*date*) Returns the day of the week from a SAS date value

YEAR(*date*) Returns the year from a SAS date value

SAS Formats

w.d	standard numeric
COMMAw.d	writes numeric values with commas and decimal points
Zw.d	print leading zeros
$w.	writes standard character data
$CHARw.	writes standard character data (including leading blanks)
$VARYINGw.	Writes character data of varying length

SAS Informats

w.d	Reads standard numeric data
datew.	Reads date values (ddmmmyy)
$w.	Reads standard character data
$VARYINGw.	Reads character data of varying length

Compliments of:

David Franklin
New Hampshire, USA
Tel/Fax +1(603) 262-9160 Cell +1(603) 275-6809
Email 100316.3451@compuserve.com
Google "Franklin SAS"
http://ourworld.compuserve.com/homepages/dfranklinuk

SAS® Cheat Sheet

SAS Procedures

PROC COMPARE <BASE=*dset*> <COMPARE=*dset*>;
  BY *variable(s)*;
  ID *variable(s)*;
  VAR *variable(s)*;

PROC DATASETS <LIBRARY=*libref*> <MEMTYPE=(*m-list*)>
  <DETAILS|NODETAILS> <KILL>
  <NOLIST>;
  APPEND BASE=*dset* <DATA=*dset*> <FORCE>;
  CHANGE *old-name$_n$=new-name$_n$* </MEMTYPE=(*m-list*);
  CONTENTS <DATA=*<libref.>member*> <DIRECTORY>
  <MEMTYPE=(*m-list*)> <NODS>
  <VARNUM> <NOPRINT> <OUT=*dset*>;
  COPY OUT=*libref* <IN=*libref*> <MEMTYPE=(*m-list*)>
  <MOVE>;
    EXCLUDE *member-list* </MEMTYPE=*mtype*>;
    SELECT *member-list* </MEMTYPE=*mtype*>;
  DELETE *member-list* </ MEMTYPE=*mtype*>;
  MODIFY *member-name* <(<LABEL=*'data-set-label'*|' '>
  <SORTEDBY=*sort-information*>)>;
    FORMAT *variable-format-name.;*
    INDEX CREATE *variable* </<UNIQUE> <NOMISS>>;
    INDEX CREATE *index=(variable-list)* </<UNIQUE>
  <NOMISS>>;
    INDEX DELETE *index-list*;
    LABEL *variable='label-text';*
    RENAME variable$_n$=*new-variable$_n$*;
  QUIT;

where

m-list	one or more of the member types that processing should be restricted to.
member-list	list of members in the directory to process.
mtype	restricts processing to one member type.

PROC EXPORT DATA=*<libref.>dset*
  OUTFILE="*filename*" <REPLACE>;
PROC IMPORT DATAFILE="*filename*"
  OUT=*<libref.>dset* <REPLACE>;
  The following filetypes are the most commonly used and supported within *filename* by SAS:
    filename.XLS (Microsoft Excel)
    filename.TXT (tab delimited)
    filename.CSV (comma separated value)

PROC FORMAT <CNTLIN=*dset*>
  <CNTLOUT=*dset*>
  <LIBRARY=*libref<.catalog>*>;
  INVALUE *<$>name <value-range-set(s)>*;
  PICTURE *name <...value-range-set <(picture-option(s))>>;*
  VALUE *<$>name <value-range-set>*;

where

picture-options	The following options are useful: ROUND, NOEDIT, PREFIX=, FILL=

PROC FREQ <DATA=*dset*>
  <ORDER=DATA|EXTERNAL|FREQ|INTERNAL>;
  BY <DESCENDING> *var$_n$*;
  TABLES *requests </tables-options>*;

where

requests	one or more variable names joined by asterisks that specify the form of the generated tables, e.g. A*B
tables-options	Can be one or more of the following: LIST, MISSING, NOPRINT, OUT=SAS-data-set, OUTPCT, SPARSE

PROC MEANS <DATA=*dset*> <DESCENDING>
  <MISSING> <NOPRINT> <NWAY>
  <ORDER=DATA|EXTERNAL|FREQ|
  INTERNAL>
  <*statistic-list*>;
  VAR *variable-list*;
  CLASS *variable-list*;
  OUTPUT <OUT=*dset*> <*out-statistic*>;

where

statistic-list	Can be one or more of the following: N, NMISS, MIN, MAX, RANGE, MEDIAN, SUM, MEAN, VAR, STD, Q1, Q3, T
out-statistic	Specifies the statistics in the output and also names the variable(s) that contain the results.

PROC REPORT <DATA=*dset*> <HEADLINE> <HEADSKIP>
  <NOWINDOWS> <SPACING=*number*>;
  COLUMNS *<report-item$_1$, <., report-item$_n$>>*
    (*`header$_1$ ' < . `header$_n$ '> report-item(s)*);
  DEFINE *report-item / <usage> <define-options>;*
  COMPUTE <BEFORE|AFTER> *report-item;*
    LINE *<item item-format | 'text' | pointer-control>*;
    ENDCOMP;
  BREAK BEFORE|AFTER *break-variable </b-option(s)>*;
  QUIT;

where

report-item	name or alias (established in the COLUMN statement) of the data set or computed variable, or statistic to define
usage	Either ACROSS, ANALYSIS, COMPUTED, DISPLAY, GROUP, ORDER
define-options	The following options are available: FORMAT=*format*, ORDER=, SPACING=, WIDTH=, DESCENDING, FLOW, NOPRINT, CENTER, LEFT, RIGHT, COLOR=*'column-header'*
b-options	These include: SKIP, PAGE

PROC SORT <DATA=*dset*> <OUT=*dset*>
  <NODUPKEY|NODUPS>;
  BY <DESCENDING> *variable-list*;

PROC TRANSPOSE <DATA=*dset*> <OUT=*dset*>;
  BY <DESCENDING> *variable-list*;
  ID *variable*;
  VAR *variable$_1$ … variable$_n$*;

Macro Language

%DO *macro-var=start_value* %TO *end_value* <%BY *step*>;
  Executes a section of a macro repetitively based on the value of an index variable
%DO %WHILE *(expression)*;
  Executes a section of a macro repetitively while a condition is true
%DO %UNTIL *(expression)*;
  Executes a section of a macro repetitively until a condition is true
%GLOBAL *macro-variable(s)*;
  Creates macro variables that are available during the execution of an entire SAS session
%IF *expression* %THEN *action*; <%ELSE *action*;>
  Conditionally process a portion of a macro
%LENGTH *(character string | text expression)*
  Returns the length of a string
%LET *macro-variable =<value>*;
  Creates a macro variable and assigns it a value
%MACRO *m-name (<pp$_1$><…,pp$_n$><kp$_1$=value<..<kp$_n$=v>)*;
  Begins a macro definition
%MEND <*macro-name*>;
  Ends a macro definition
%SCAN(*argument$_n$<,delimiters>*)
  Search for a word that is specified by its position in a string
%SUBSTR(*argument,position<,length>*)
  Produce a substring of a character string
%UPCASE(*character string | text expression*)
  Convert values to uppercase

Macro Quoting

%QUOTE | %NRQUOTE and %BQUOTE | %NRBQUOTE
  Mask special characters and mnemonic operators in a resolved value at macro execution
%STR | %NRSTR
  Mask special characters and mnemonic operators in constant text at macro compilation
%SUPERQ
  Masks special characters/mnemonic operators at macro execution but prevents further resolution of the value.

Compliments of:

David Franklin
New Hampshire, USA
Tel/Fax +1(603) 262-9160 Cell +1(603) 275-6809
Email 100316.3451@compuserve.com
Google "Franklin SAS"
http://ourworld.compuserve.com/homepages/dfranklinuk

Regression: Predict Continuous Data

Predict how a dependent variable (output, *t*) changes when any of the independent variables (inputs, or features, *x*) change, for example, how house prices change as a function of neighborhood and size, or how time spent on a web page varies as a function of the number of ads and content type. Training data has *N* samples and *D* features.

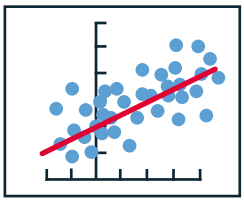
Linear Model $O(ND^2)$

Solves problems of the form:

$$y = w_0 + w_1 x_1 + \ldots + w_d x_d$$

with predicted value *y*, features, $\boldsymbol{x}$, and fitted weights $\boldsymbol{w}$.
Solved by minimizing "least square error", E_D:

$$E_D(\mathbf{w}, \mathbf{x}) = \sum_{n=0}^{N-1} (t_n - y_n)^2$$



On fitted models, access $\boldsymbol{w}$ as `model.coef_` and w_0 as `model.intercept_`.
Gotchas: Features must be uncorrelated, use `decomposition.PCA()`.
Code: `linear_model.LinearRegression()` if less than 100k samples, or see SGD.

Ridge $O(ND^2)$

When to use it: Less than 100k samples, noisy outputs.
How it works: Linear model that limits the size of the weights. Prevents overfitting by increasing bias. Minimizes *E* instead of E_D, where second term is called "L2 norm":

$$E(\mathbf{w}, \mathbf{x}) = E_D(\mathbf{w}, \mathbf{x}) + \alpha \frac{1}{2} \sum_{d=0}^{D-1} w_d^2$$

Code: `linear_model.Ridge(alpha)`
- **alpha**: Regularization strength, **alpha** > 0, corresponds to 1/*C* in other models. Increase if noisy samples.

Lasso $O(ND^2)$

When to use it: Less than 100k samples, only some features should be important.
How it works: Linear model that forces small weights to be zero. Minimizes *E* instead of E_D, where the second term is called "L1 norm":

$$E(\mathbf{w}, \mathbf{x}) = E_D(\mathbf{w}, \mathbf{x}) + \alpha \frac{1}{2} \sum_{d=1}^{D-1} |w_d|$$

Code: `linear_model.Lasso(alpha)`
- **alpha**: Regularization strength, **alpha** > 0, corresponds to 1/*C* in other models. Increase if noisy samples.

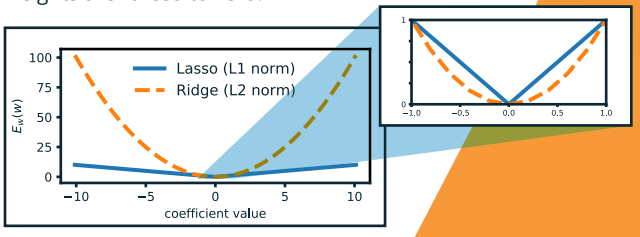
Tip: Use with `feature_selection.SelectFromModel` as a transformation stage to select features with non-zero weights.

Ridge vs. Lasso – Shape of E_w

With Ridge and Lasso, the error to minimize *E* has an extra component E_w:

$$E(\mathbf{w}, \mathbf{x}) = E_D(\mathbf{w}, \mathbf{x}) + \alpha E_W(\mathbf{w})$$

Lasso produces sparse models because small weights are forced to zero.



Nonlinear Transformations

When to use them: "Straight line" not sufficient, for example, predicting temperature has a function of time of day.
How it works: "Reword" a nonlinear model in linear terms using *nonlinear basis functions*, $\Phi_j(x)$, so we can use linear model machinery to solve nonlinear problems. Linear model becomes:

$$y(\mathbf{w}, \mathbf{x}) = w_0 \phi_0(\mathbf{x}) + w_1 \phi_1(\mathbf{x}) + \ldots + w_j \phi_j(\mathbf{x})$$

Polynomial Expansion of Order P: E.g. A 2nd order polynomial two-feature model: $y(\mathbf{w}, \mathbf{x}) = (w_0 + w_1 x_1 + w_2 x_2)^2$
Becomes a model with these 6 basis functions:

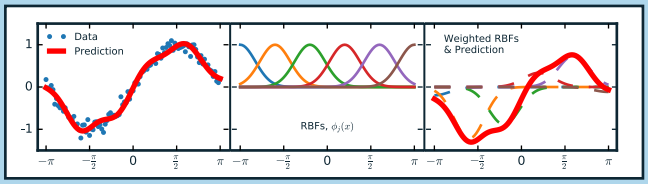
$$\phi_0(\mathbf{x}) = 1,\ \phi_1(\mathbf{x}) = x_1,\ \phi_2(\mathbf{x}) = x_2,\ \phi_3(\mathbf{x}) = x_1^2,\ \phi_4(\mathbf{x}) = x_1 x_2,\ \phi_5(\mathbf{x}) = x_2^2,$$

Gotchas: Same feature affects many different coefficients, so an outlier can have a big global effect. Number of basis functions grows very quickly, $O((P+1)^{(D+1)})$.

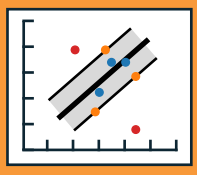
Code:
```
poly=preprocessing.PolynomialFeatures(degree)
x_poly = poly.fit_transform(x)
```

Radial Basis Functions (RBF): Local, Gaussian-shaped functions, defined by centers and width. Turns one feature into *P* features.

Code: `metrics.pairwise.rbf_kernel(x, centers, gamma)`

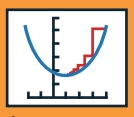


Support Vector Regressor ~$O(N^2D)$



When to use it: Many important features, more features than samples, nonlinear problem.
How it works: Find a function such that training points fit within a "tube" of acceptable error, with some tolerance towards points that are outside the tube.
Gotchas: Must scale inputs, see **StandardScalar** and **RobustScalar**.
Code: Start with `svm.LinearSVR(epsilon, C=1)`.
Make *C* smaller if lots of noisy observations (*C* = 1/*α*, small *C* means more regularization).
If LinearSVR doesn't work, use `svm.SVR(kernel='rbf', gamma)`.

Stochastic Gradient Descent (SGD) Regressor



When to use it: Fit is too slow with other estimators.
How it works: "Online" method, learns the weights in batches, with a subset of the data each time. Pair with manual basis function expansion to train nonlinear models on really large datasets.
Code: `linear_model.SGDRegressor()` and `partial_fit()` method.

Performance Metrics in `sklearn.metrics`

`mean_squared_error`: Smaller is better. Puts large weight on outliers.

$$\frac{1}{N} \sum_{n=0}^{N-1} (t_n - y_n)^2$$

`r2_score`: Coefficient of determination. Best score is 1.0. Proportion of explained variance. Default for `model.score(x, t)`.

$$1 - \frac{\sum_{n=0}^{N-1} (t_n - y_n)^2}{\sum_{n=0}^{N-1} (t_n - \text{mean}(\mathbf{t}))^2}$$

`mean_absolute_error`: Smaller is better. Uses same scale as the data.

$$\frac{1}{N} \sum_{n=0}^{N-1} |t_n - y_n|$$

`median_absolute_error`: Robust to outliers.

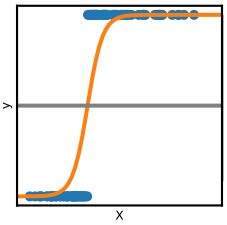
$$\text{median}(|t_0 - y_0|, \ldots, |t_n, y_n|)$$

ENTHOUGHT

Classification: Predict Categorical Data

Predict the class, or label (*t*), of a sample based on its features (*x*). Examples: Recognize hand-written digits, or mark email as spam. In scikit-learn, labels are represented as integers and get expanded internally into matrices of binary choices between unique integer labels. Use `class_weight='balanced'` in most models to adjust for unbalanced datasets (more training data from one class than others). Training data has *N* samples and *D* features.

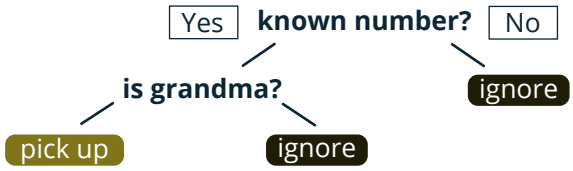
Logistic Regression $O(ND^2)$



When to use it: Need to understand contributions of features. Fast to train, easy to interpret.
How it works: Fits an s-shaped function (logistic function), which is continuous but has a steep transition between the two classes, and assigns class based on sign.
Gotchas: Inputs must be scaled and uncorrelated.
Code: `linear_model.LogisticRegression(C, solver)`.
- `penalty='l1'` to use estimator for feature selection.
- `solver='liblinear'` for small datasets or L1 penalty, `'lbfgs'`, `'sag'` or `'newton-cg'` for multi-class problems and large datasets, and `'sag'` for very large datasets.

Decision Tree $O(NDlog(N))$

When to use it: Need to understand prediction decisions. Data has both continuous and categorical features. No scaling needed.
How it works: Chain binary decisions on increasingly smaller subsets of data. Deeper trees have more complex decision rules and a better fit.
Gotchas: Very often overfits. Consider doing dimensionality reduction beforehand. *N* must double with each extra level.
Code: `tree.DecisionTreeClassifier(max_depth)`. Start with `max_depth=3`, then increase. Use `tree.export_graphviz` to visualize tree.



Ensemble Methods

When to use them: No single estimator gave satisfying results.
How they work: "Wisdom of the crowd". Combines predictions of multiple weak, biased estimators to create a better one. Two types: *averaging methods* build many estimators and average predictions; in *boosting methods* each new estimator tries to improve the previous one.
Gotchas: Hard to generate the perfect mix of estimators.
Code: All in `ensemble` module.

Averaging estimators:
- `RandomForestClassifier(max_features)`
- `ExtraTreesClassifier(max_features)`

Start with these, but always cross-validate:
- `max_features=sqrt(n_features)`
- `max_depth=None`
- `min_samples_split=1`

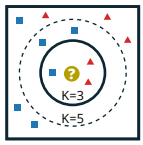
Boosting estimator:
- `AdaBoostClassifier()`
- `GradientBoostingClassifier()`

All:
- `Parallelize with n_jobs=-1`
  Increasing `n_estimators` is better, but slower

Support Vector Classifier $O(ND^2)$ to $O(ND^3)$

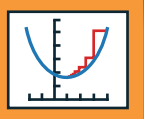
When to use it: Large number of features. Slightly more features than samples.
How it works: Maximize distance between classes in high-dimensional space, i.e., "maximum margin classifier".
Gotchas: Scale your data.
Code: `svm.SVC(kernel, C=1)`.
Make C smaller if lots of noisy samples.
If accuracy is important set `kernel='rbf'`.
If fast training is important, use `svm.LinearSVC()`.

Neighbor Classifiers $O(DlogN)$ to $O(DN)$



When to use them: Large datasets. Very irregular decision boundary.
How it works: Predict class by majority vote from nearby data.
Gotchas: Efficiency comes at the cost of also having high variance.
Code: `neighbors.KNeighborsClassifier(n_neighbors)`.
- Use `RadiusNeighborsClassifier()` for unbalanced data and *D* not too large.
- Try `weights='uniform'` and `'distance'`.

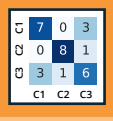
Stochastic Gradient Descent (SGD) Classifier



When to use it: Very large *N* and *D*, e.g., 10^5 samples and 10^5 features.
How it works: "Online" method, learns the weights in batches.
Gotchas: Data must be scaled.
Code: `linear_model.SGDClassifier(loss, alpha, n_iter)` and `partial_fit()` method.
- Use `n_iter=np.ceil(10**6/n_samples)`. `loss='hinge'` gives SVC, `'log'` gives logistic regression.

Performance Metrics in `sklearn.metrics`

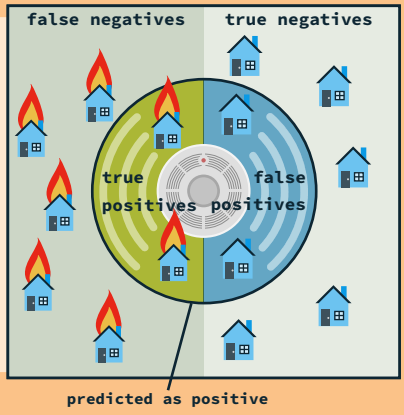
They take targets, `t`, and predicted classes, `y`, as arguments.
There's more than one way to be wrong. A fire alarm that always goes off is annoying, one that never goes off is costly.
- `confusion_matrix`: Explore how model confuses classes. Visualize with `seaborn.heatmap`.



- `accuracy_score` (default for `model.score`): Fraction correctly predicted. Meaningless if samples are unbalanced. (TP + TN) / Total

- `recall_score`: Fraction of predicted fire when there's actually fire. TP / (TP + FN)

- `precision_score`: Fraction of *correctly* predicted fire of all cases where fire is predicted. P predicted as P. TP / (TP + FP)



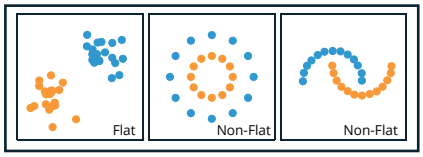
ENTHOUGHT

Take your Machine Learning skills to the next level! Register at www.enthought.com/python-for-data-science-training

Clustering: Unsupervised Learning

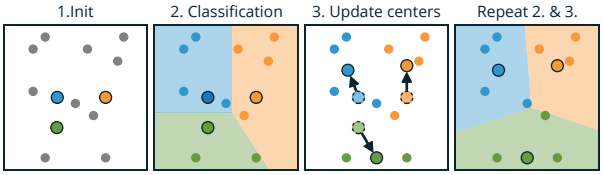
Predict the underlying structure in features, without the use of targets or labels. Splits samples into groups called "clusters". With no targets, models are trained by minimizing some definition of "distance" within a cluster. Data has *N* samples, *D* features, and the model discovers *k* clusters. Models can be used for prediction or for transformation, by reducing *D* features into one with *k* unique values.

Some models expect geometries that are "flat", or roughly spherical. Clusters with complicated shapes like rings or lines are not flat, and will not work in those models.



K-Means $O(kN)$

When to use it: Scales well. Works best on a small number of flat clusters. For large sample sizes, substitute MiniBatchKMeans.
How it works: Assigns samples to nearest of *k* cluster centers, then moves the centers to minize the average distance between centers and samples.
Gotchas: The K-Means algorithm used by scikit-learn is sensitive to initial location of the centers. Performs poorly on complex, non-flat shapes.
Code: `cluster.KMeans(n_clusters)`. `n_jobs=-1` to parallelize.



Mean Shift $O(N \log N)$

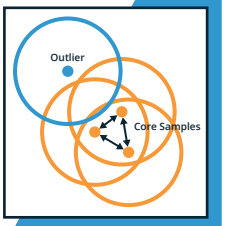
When to use it: Non-flat geometries. Unknown number of clusters. Need to guarantee convergence.
How it works: Finds local maxima given a window size.
Gotchas: Accuracy strongly tied to selecting correct window.
Code: `cluster.MeanShift(bandwith)`. Set bandwidth manually to small value for large dataset. Estimating it is $O(N^2)$ and can be the bottleneck.

Affinity Propagation $O(N^2)$

When to use it: Unknown number of clusters. Need to specify own similarity metric (`affinity` argument).
How it works: Finds data points which maximize similarity within cluster while minimizing similarity with data outside of cluster.
Gotchas: $O(N^2)$ memory use. Accuracy tied to `damping`.
Code: `cluster.AffinityPropagation(preference, damping)`
- `preference`: Negative. Controls the number of clusters. Explore on log scale.
- `damping`: 0.5 to 1.

DBSCAN $O(N^2)$

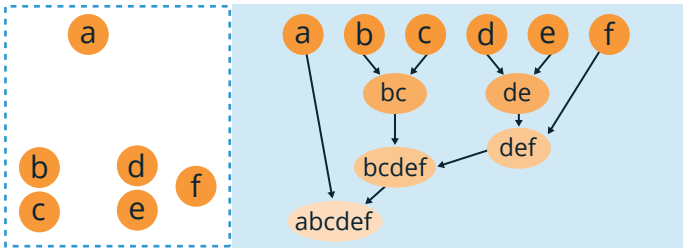
When to use it: Very non-flat geometries. Very uneven clusters.
How it works: Clusters are contiguous areas with high data density. Bounds of clusters are found using graph connectivity.
Gotchas: $O(N^2)$ memory use. Not deterministic at cluster boundaries.
Code: `cluster.DBSCAN(min_samples, eps, metric)`
- Higher `min_samples`, or lower `eps` requires higher density to form a cluster.



Agglomerative Clustering $O(N^2 \log N)$

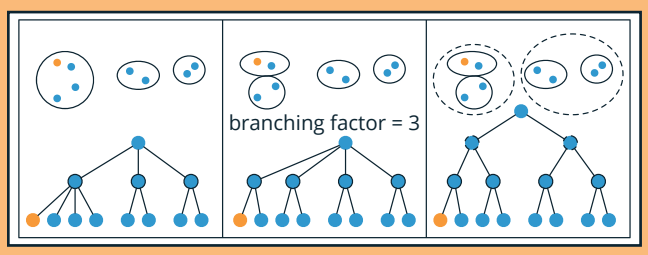
When to use it: Need a flexible definition of distance (e.g. Levenshtein).
How it works: Defines all observations as unique clusters, then merges the closest ones iteratively.
Gotchas: Worst time complexity. "Rich get richer" behavior.
Code: `cluster.AgglomerativeClustering(linkage, affinity, connectivity)`. Set `linkage` criteria for merging:
- `'ward'`: minimize sum of square differences. Minimizes variance. Gives most regular cluster size.
- `'complete'`: minimize max distance between sample pairs.
- `'average'`: minimize average distance between all sample pairs Yields uneven cluster sizes.

`affinity`: defines type of distances. `'l1'` for sparse features, e.g., text; `'cosine'` is invariant to scaling.
`connectivity`: provides extra constraints about which nodes can be merged, e.g., `neighbors.kneighbors_graph`.



BIRCH $O(kN)$

When to use it: Large number of observations and small number of features.
How it works: Builds a balanced tree of groups of data, then clusters those groups instead of the raw data.
Gotchas: Performs poorly with large number of features.
Code: `cluster.Birch(threshold, branching_factor, n_clusters)`



Performance Metrics in `sklearn.metrics`

The metrics do not take into account the exact class values, only their separation. Score is based on *ground truth*, (targets), if available, or to a measure of similarity within class, and difference across classes.
Needs ground truth:
- `adjusted_rand_score`: -1 to 1 (best). 0 is random classes. Measures similarity. Related to accuracy (% correct).
- `adjusted_mutual_info_score`: 0 to 1 (best). 0 is random classes. 10x slower than `adjusted_rand_score`. Measures agreement.
- `homogeneity_completeness_v_measure`: 0 to 1 (best). `homogeneity`: each cluster only contains members of one class; `completeness`: all members of a class are in the same cluster: and, `v_measure_score`: the harmonic mean of both. Not normalized for random labeling. Doesn't need ground truth:
- `silhouette_score`: -1 to 1 (best). 0 means overlapping clusters. Based on distance to samples in same cluster and distance to next nearest cluster.

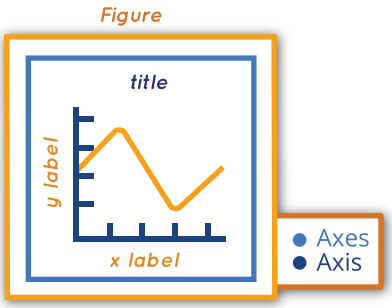
ENTHOUGHT

Plotting with Pandas Series and DataFrames

Pandas uses Matplotlib to generate figures. Once a figure is generated with Pandas, all of Matplotlib's functions can be used to modify the title, labels, legend, etc. In a Jupyter notebook, all plotting calls for a given plot should be in the same cell.

Parts of a Figure

An Axes object is what we think of as a "plot". It has a title and two Axis objects that define data limits. Each Axis can have a label. There can be multiple Axes objects in a Figure.



Setup

Import packages:

```
> import pandas as pd
> import matplotlib.pyplot as plt
```

Execute this at IPython prompt to display figures in new windows:

```
> %matplotlib
```

Use this in Jupyter notebooks to display static images inline:

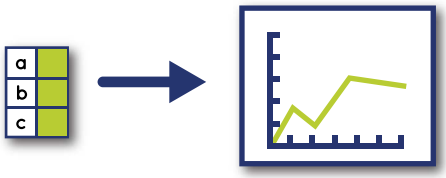
```
> %matplotlib inline
```

Use this in Jupyter notebooks to display zoom-able images inline:

```
> %matplotlib notebook
```

Plotting with Pandas Objects

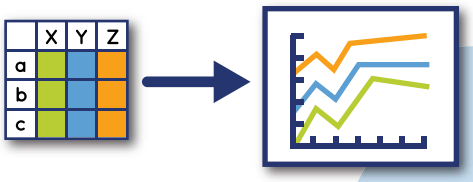
Series



With a Series, Pandas plots values against the index:

```
> ax = s.plot()
```

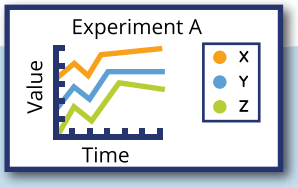
Dataframe



With a DataFrame, Pandas creates one line per column:

```
> ax = df.plot()
```

Labels



Use Matplotlib to override or add annotations:

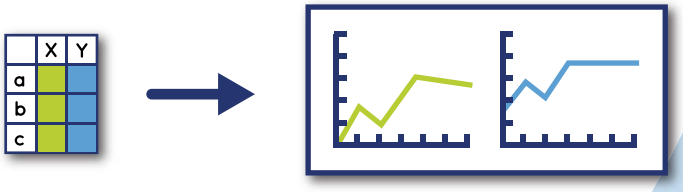
```
> ax.set_xlabel('Time')
> ax.set_ylabel('Value')
> ax.set_title('Experiment A')
```

Pass labels if you want to override the column names and set the legend location:

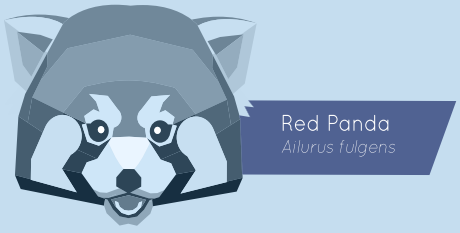
```
> ax.legend(labels, loc='best')
```

When plotting the results of complex manipulations with **groupby**, it's often useful to **stack/unstack** the resulting DataFrame to fit the one-line-per-column assumption (see Data Structures cheatsheet).

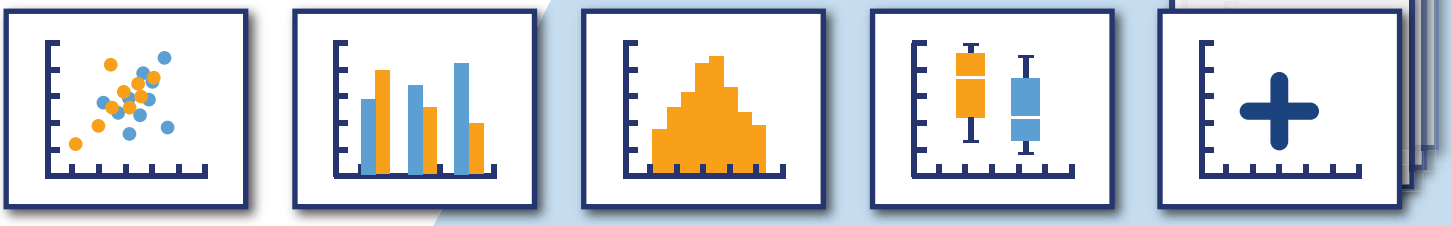
Useful Arguments to **plot**



- **subplots=True**: one subplot per column, instead of one line
- **figsize**: set figure size, in inches
- **x** and **y**: plot one column against another



Kinds of Plots



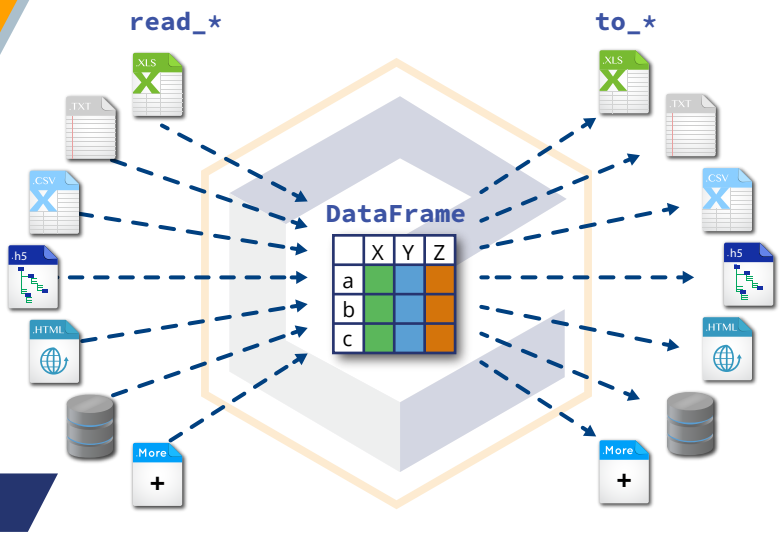
`df.plot(kind='scatter')` `df.plot(kind='bar')` `df.plot(kind='hist')` `df.boxplot()`

ENTHOUGHT

Take your Pandas skills to the next level! Register at www.enthought.com/pandas-master-class

Reading and Writing Data with Pandas

Methods to read data are all named **`pd.read_*`** where `*` is the file type. Series and DataFrames can be saved to disk using their **`to_*`** method.



Usage Patterns

- Use **`pd.read_clipboard()`** for one-off data extractions.
- Use the other **`pd.read_*`** methods in scripts for repeatable analyses.

Reading Text Files into a DataFrame

Colors highlight how different arguments map from the data file to a DataFrame.

```
# Historical_data.csv
Date, Cs, Rd
2005-01-03, 64.78, -
2005-01-04, 63.79, 201.4
2005-01-05, 64.46, 193.45
...
Data from Lab Z.
  Recorded by Agent E
```

```
>>> read_table(
        'historical_data.csv',
         sep=',',
         header=1,
         skiprows=1,
         skipfooter=2,
         index_col=0,
         parse_dates=True,
         na_values=['-'])
```

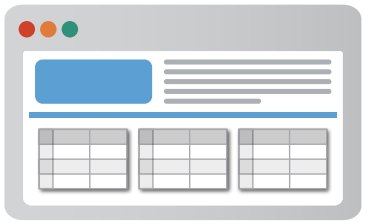
Date	Cs	Rd

Other arguments:
- **`names`**: set or override column names
- **`parse_dates`**: accepts multiple argument types, see on the right
- **`converters`**: manually process each element in a column
- **`comment`**: character indicating commented line
- **`chunksize`**: read only a certain number of rows each time

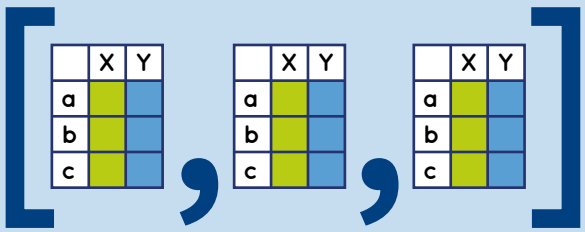
Possible values of **`parse_dates`**:
- **`[0, 2]`**: Parse columns 0 and 2 as separate dates
- **`[[0, 2]]`**: Group columns 0 and 2 and parse as single date
- **`{'Date': [0, 2]}`**: Group columns 0 and 2, parse as single date in a column named Date.

Dates are parsed *after* the **`converters`** have been applied.

Parsing Tables from the Web



```
>>> df_list = read_html(url)
```



Writing Data Structures to Disk

Writing data structures to disk:

```
> s_df.to_csv(filename)
> s_df.to_excel(filename)
```

Write multiple DataFrames to single Excel file:

```
> writer = pd.ExcelWriter(filename)
> df1.to_excel(writer, sheet_name='First')
> df2.to_excel(writer, sheet_name='Second')
> writer.save()
```

From and To a Database

Read, using SQLAlchemy. Supports multiple databases:

```
> from sqlalchemy import create_engine
> engine = create_engine(database_url)
> conn = engine.connect()
> df = pd.read_sql(query_str_or_table_name, conn)
```

Write:

```
> df.to_sql(table_name, conn)
```

ENTHOUGHT

Computation with Series and DataFrames

Pandas objects do not behave exactly like Numpy arrays. They follow three main rules *(see on the right)*. Aligning objects on the index (or columns) before calculations might be the most important difference. There are built-in methods for most common statistical operations, such as **mean** or **sum**, and they apply across one-dimension at a time. To apply custom functions, use one of three methods to do tablewise (**pipe**), row or column-wise (**apply**) or elementwise (**applymap**) operations.

The **3 Rules** of Binary Operations

Rule 1:

Operations between multiple Pandas objects implement auto-alignment based on index first.

Rule 2:

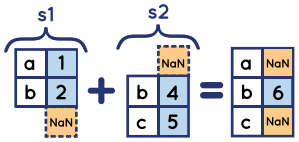
Mathematical operators (+ - * / exp, log, ...) apply element by element, on the values.

Rule 3:

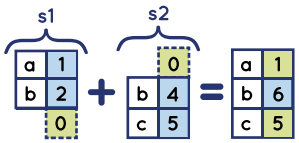
Reduction operations (mean, std, skew, kurt, sum, prod, ...) are applied column by column by default.

Rule 1: Alignment First

```
> s1 + s2
```

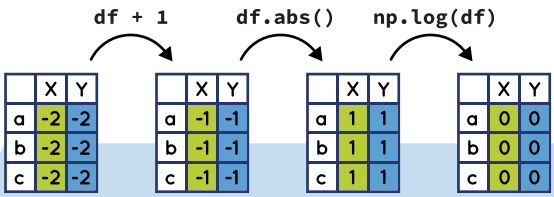



```
> s1.add(s2, fill_value=0)
```



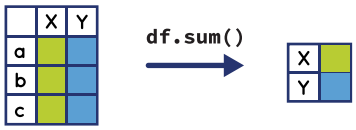
Use **add**, **sub**, **mul**, **div**, to set fill value.

Rule 2: Element-By-Element Mathematical Operations



Rule 3: Reduction Operations

```
>>> df.sum()  ⟶  Series
```



Operates across rows by default (**axis=0**, or **axis='rows'**).
Operate across columns with **axis=1** or **axis='columns'**.

Reduction functions

count:	Number of non-null observations
sum:	Sum of values
mean:	Mean of values
mad:	Mean absolute deviation
median:	Arithmetic median of values
min:	Minimum
max:	Maximum
mode:	Mode
prod:	Product of values
std:	Bessel-corrected sample standard deviation
var:	Unbiased variance
sem:	Standard error of the mean
skew:	Sample skewness (3rd moment)
kurt:	Sample kurtosis (4th moment)
quantile:	Sample quantile (Value at %)
value_counts:	Count of unique values

Apply a Function to Each Value

Apply a function to each value in a Series or DataFrame

```
s.apply(value_to_value)     ⟶ Series
df.applymap(value_to_value) ⟶ DataFrame
```

Apply a Function to Each Series

Apply **series_to_*** function to every column by default (across rows):

```
df.apply(series_to_series)  ⟶ DataFrame
df.apply(series_to_value)   ⟶ Series
```

To apply the function to every row (across columns), set **axis=1**:

```
df.apply(series_to_series, axis=1)
```

Apply a Function to a DataFrame

Apply a function that receives a DataFrame and returns a DataFrame, a Series, or a single value:

```
df.pipe(df_to_df)     ⟶ DataFrame
df.pipe(df_to_series) ⟶ Series
df.pipe(df_to_value)  ⟶ Value
```

What Happens with Missing Values?

Missing values are represented by **NaN** (not a number) or **NaT** (not a time).

- They propagate in operations across Pandas objects (**1 + NaN ⟶ NaN**).
- They are ignored in a "sensible" way in computations, they equal 0 in **sum**, they're ignored in **mean**, etc.
- They stay **NaN** with mathematical operations (**np.log(NaN) ⟶ NaN**).

ENTHOUGHT

Split / Apply / Combine with DataFrames

1. Split the data based on some criteria.
2. Apply a function to each group to aggregate, transform, or filter.
3. Combine the results.

The apply and combine steps are typically done together in Pandas.

Split: Group By

Group by a single column:

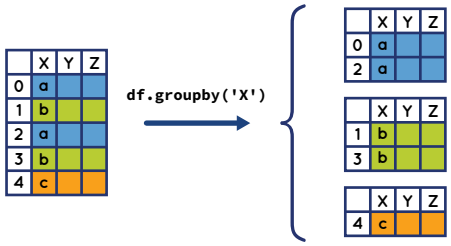
```
> g = df.groupby(col_name)
```

Grouping with list of column names creates DataFrame with MultiIndex. (see "Reshaping DataFrames and Pivot Tables" cheatsheet):

```
> g = df.groupby(list_col_names)
```

Pass a function to group based on the index:

```
> g = df.groupby(function)
```



Apply/Combine: General Tool: apply

More general than **agg**, **transform**, and **filter**. Can aggregate, transform or filter. The resulting dimensions can change, for example:

```
> g.apply(lambda x: x.describe())
```

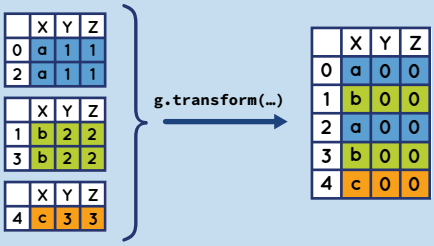
Apply/Combine: Transformation

The shape and the index do not change.

```
> g.transform(df_to_df)
```

Example, normalization:

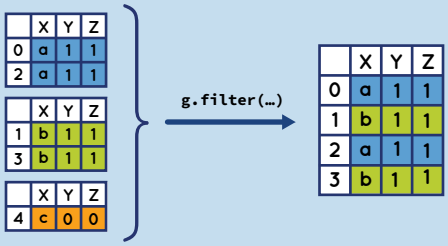
```
> def normalize(grp):
.      return (grp - grp.mean()) / grp.var()
> g.transform(normalize)
```



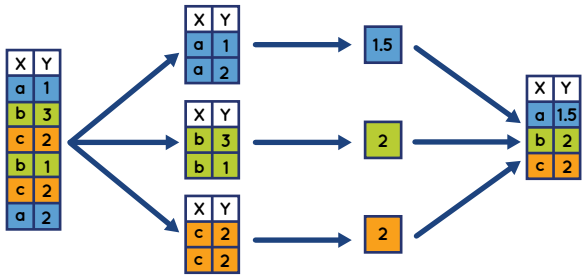
Apply/Combine: Filtering

Returns a group only if condition is true.

```
> g.filter(lambda x: len(x)>1)
```



Split/Apply/Combine



Split	Apply	Combine
• Groupby • Window Functions	• Apply • Group-specific transformations • Aggregation • Group-specific Filtering	

Split: What's a GroupBy Object?

It keeps track of which rows are part of which group.

```
> g.groups
```
⟶ Dictionary, where keys are group names, and values are indices of rows in a given group.

It is iterable:

```
> for group, sub_df in g:
      ...
```

Apply/Combine: Aggregation

Perform computations on each group. The shape changes; the categories in the grouping columns become the index. Can use built-in aggregation methods: **mean**, **sum**, **size**, **count**, **std**, **var**, **sem**, **describe**, **first**, **last**, **nth**, **min**, **max**, for example:

```
> g.mean()
```

… or aggregate using custom function:

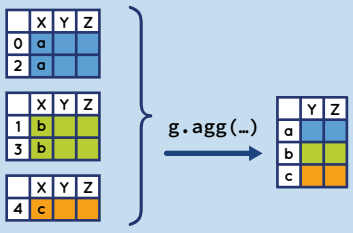
```
> g.agg(series_to_value)
```

… or aggregate with multiple functions at once:

```
> g.agg([s_to_v1, s_to_v2])
```

… or use different functions on different columns.

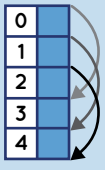
```
> g.agg({'Y': s_to_v1, 'Z': s_to_v2})
```



ENTHOUGHT

Other Groupby-Like Operations: Window Functions

- **resample**, **rolling**, and **ewm** (exponential weighted function) methods behave like GroupBy objects. They keep track of which row is in which "group". Results must be aggregated with **sum**, **mean**, **count**, etc. (see Aggregation).
- **resample** is often used before **rolling**, **expanding**, and **ewm** when using a DateTime index.



Manipulating Dates and Times

Use a Datetime index for easy time-based indexing and slicing, as well as for powerful resampling and data alignment.

Pandas makes a distinction between timestamps, called **`Datetime`** objects, and time spans, called **`Period`** objects.

Converting Objects to Time Objects

Convert different types, for example strings, lists, or arrays to Datetime with:

```
> pd.to_datetime(value)
```

Convert timestamps to time spans: set period "duration" with frequency offset (see below).

```
> date_obj.to_period(freq=freq_offset)
```

Creating Ranges of Timestamps

```
> pd.date_range(start=None, end=None,
                periods=None, freq=offset,
                tz='Europe/London')
```

Specify either a start or end date, or both. Set number of "steps" with **`periods`**. Set "step size" with **`freq`**; see "Frequency offsets" for acceptable values. Specify time zones with **`tz`**.

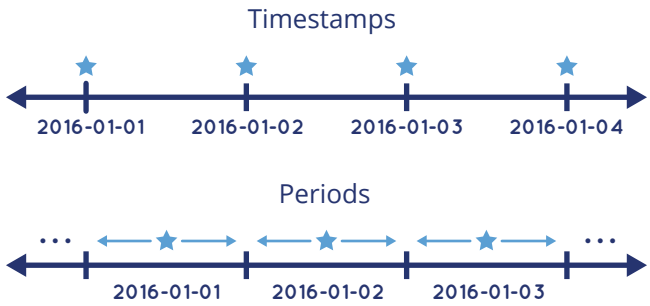
Frequency Offsets

Used by **`date_range`**, **`period_range`** and **`resample`**:

- B: Business day
- D: Calendar day
- W: Weekly
- M: Month end
- MS: Month start
- BM: Business month end
- Q: Quarter end
- A: Year end
- AS: Year start
- H: Hourly
- T, min: Minutely
- S: Secondly
- L, ms: Milliseconds
- U, us: Microseconds
- N: Nanoseconds

For more:
Lookup "Pandas Offset Aliases" or check out **`pandas.tseries.offsets`**, and **`pandas.tseries.holiday`** modules.

Timestamps vs Periods



Save Yourself Some Pain: Use ISO 8601 Format

When entering dates, to be consistent and to lower the risk of error or confusion, use ISO format YYYY-MM-DD:

```
>>> pd.to_datetime('12/01/2000')  # 1st December
Timestamp('2000-12-01 00:00:00')
>>> pd.to_datetime('13/01/2000')  # 13th January!
Timestamp('2000-01-13 00:00:00')
>>> pd.to_datetime('2000-01-13')  # 13th January
Timestamp('2000-01-13 00:00:00')
```

Creating Ranges or Periods

```
> pd.period_range(start=None, end=None,
                  periods=None, freq=offset)
```

Resampling

```
> s_df.resample(freq_offset).mean()
```

`resample` returns a groupby-like object that must be aggregated with **`mean`**, **`sum`**, **`std`**, **`apply`**, etc. (See also the Split-Apply-Combine cheat sheet.)

Vectorized String Operations

Pandas implements vectorized string operations named after Python's string methods. Access them through the **`str`** attribute of string Series

Some String Methods

```
> s.str.lower()
> s.str.isupper()
> s.str.len()
> s.str.strip()
> s.str.normalize()
```

and more...

Index by character position:

```
> s.str[0]
```

`True` if regular expression pattern or string in Series:

```
> s.str.contains(str_or_pattern)
```

Splitting and Replacing

`split` returns a Series of lists:

```
> s.str.split()
```

Access an element of each list with **`get:`**

```
> s.str.split(char).str.get(1)
```

Return a DataFrame instead of a list:

```
> s.str.split(expand=True)
```

Find and replace with string or regular expressions:

```
> s.str.replace(str_or_regex, new)
> s.str.extract(regex)
> s.str.findall(regex)
```

ENTHOUGHT

Pandas Data Structures: Series and DataFrames

A Series, **s**, maps an index to values. It is:
- Like an ordered dictionary
- A Numpy array with row labels and a name

A DataFrame, **df**, maps index and column labels to values. It is:
- Like a dictionary of Series (columns) sharing the same index
- A 2D Numpy array with row and column labels

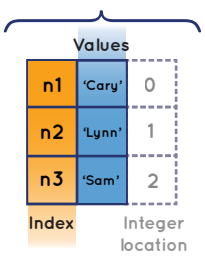
s_df applies to both Series and DataFrames.
Assume that manipulations of Pandas object return copies.

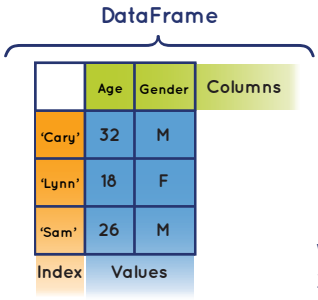
Creating Series and DataFrames

Series

```
> pd.Series(values, index=index,
            name=name)
> pd.Series({'idx1': val1, 'idx2': val2}
```

Where **values**, **index**, and **name** are sequences or arrays.





DataFrame

```
> pd.DataFrame(values, index=index,
                columns=col_names)
> pd.DataFrame({'col1': series1_or_seq,
                'col2': series2_or_seq})
```

Where **values** is a sequence of sequences or a 2D array

Manipulating Series and DataFrames

Manipulating **Columns**

```
df.rename(columns={old_name: new_name})   Renames column
df.drop(name_or_names, axis='columns')    Drops column name
```

Manipulating **Index**

```
s_df.reindex(new_index)                   Conform to new index
s_df.drop(labels_to_drop)                 Drops index labels
s_df.rename(index={old_label: new_label}) Renames index labels
s_df.reset_index()                        Drops index, replaces with Range index
s_df.sort_index()                         Sorts index labels
df.set_index(column_name_or_names)
```

Manipulating **Values**

All row values and the index will follow:

```
df.sort_values(col_name, ascending=True)
df.sort_values(['X','Y'], ascending=[False, True])
```

Important Attributes and Methods

```
s_df.index          Array-like row labels
df.columns          Array-like column labels
s_df.values         Numpy array, data
s_df.shape          (n_rows, m_cols)
s.dtype, df.dtypes  Type of Series, of each column
len(s_df)           Number of rows

s_df.head() and s_df.tail()   First/last rows
s.unique()          Series of unique values
s_df.describe()     Summary stats
df.info()           Memory usage
```

Indexing and Slicing

Use these attributes on Series and DataFrames for indexing, slicing, and assignments:

```
s_df.loc[]          Refers only to the index labels
s_df.iloc[]         Refers only to the integer location,
                    similar to lists or Numpy arrays

s_df.xs(key, level) Select rows with label key in level
                    level of an object with MultiIndex.
```

Masking and Boolean Indexing

Create masks with, for example, comparisons

```
mask = df['X'] < 0
```

Or **isin**, for membership mask

```
mask = df['X'].isin(list_valid_values)
```

Use masks for indexing (must use **loc**)

```
df.loc[mask] = 0
```

Combine multiple masks with bitwise operators (and (**&**), or (**|**), xor (**^**), not (**~**)) and group them with parentheses:

```
mask = (df['X'] < 0) & (df['Y'] == 0)
```

Common Indexing and Slicing Patterns

rows and **cols** can be values, lists, Series or masks.

```
s_df.loc[rows]          Some rows (all columns in a DataFrame)
df.loc[:, cols_list]    All rows, some columns
df.loc[rows, cols]      Subset of rows and columns
s_df.loc[mask]          Boolean mask of rows (all columns)
df.loc[mask, cols]      Boolean mask of rows, some columns
```

Using [ ] on Series and DataFrames

On Series, [ ] refers to the index labels, or to a slice

```
s['a']      Value
s[:2]       Series, first 2 rows
```

On DataFrames, [ ] refers to columns labels:

```
df['X']     Series
df[['X', 'Y']]  DataFrame

df['new_or_old_col'] = series_or_array
```

EXCEPT! with a slice or mask.

```
df[:2]      DataFrame, first 2 rows
df[mask]    DataFrame, rows where mask is True
```

NEVER CHAIN BRACKETS!

✗ `> df[mask]['X'] = 1`
SettingWithCopyWarning

✓ `> df.loc[mask , 'X'] = 1`

ENTHOUGHT

Combining DataFrames

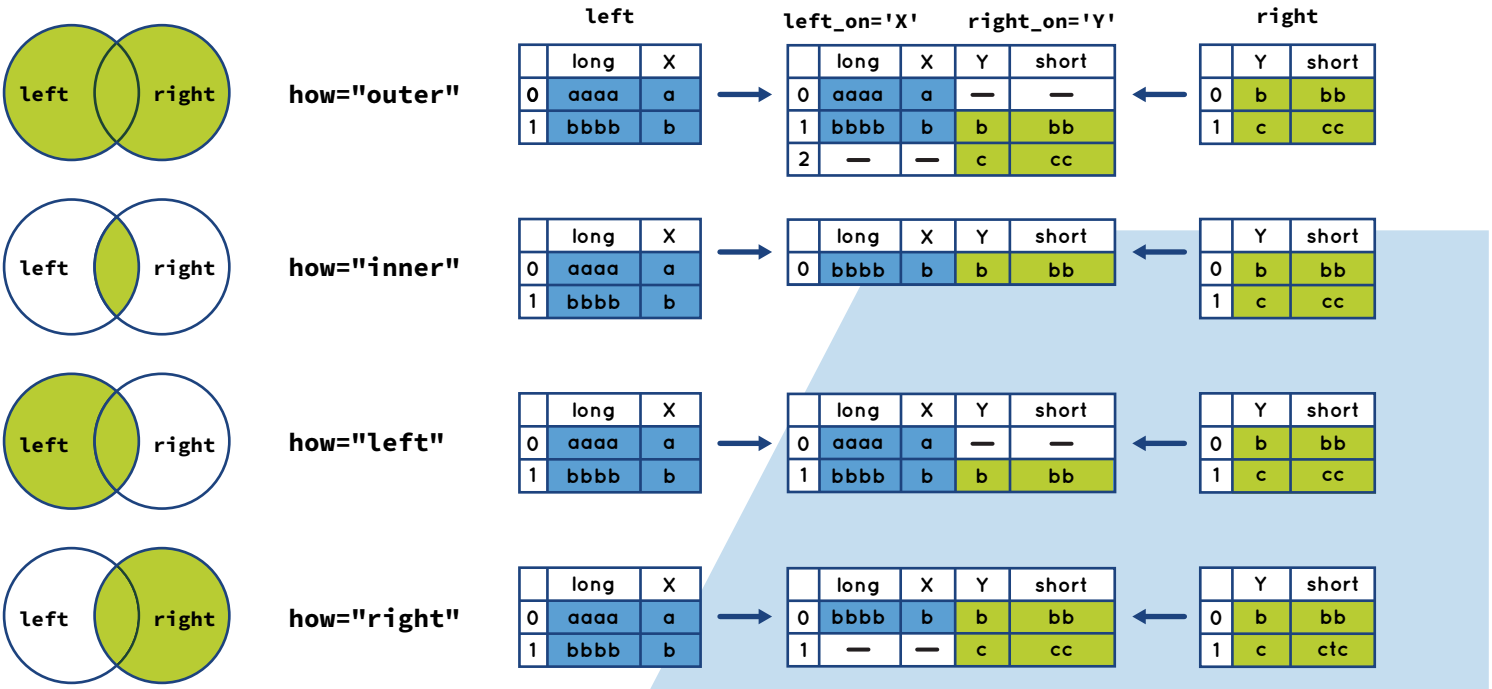
Tools for combining Series and DataFrames together, with SQL-type joins and concatenation. Use join if merging on indices, otherwise use merge.

Merge on Column Values

```
> pd.merge(left, right, how='inner', on='id')
```

Ignores index, unless **on=None**. See value of **how** below.
Use **on** if merging on same column in both DataFrames, otherwise use **left_on**, **right_on**.

Merge Types: The **how** Keyword



Concatenating DataFrames

```
> pd.concat(df_list)
```

"Stacks" DataFrames on top of each other.
Set **ignore_index=True**, to replace index with **RangeIndex**.
Note: Faster than repeated **df.append(other_df)**.

Join on Index

```
> df.join(other)
```

Merge DataFrames on index. Set **on=keys** to join on index of **df** and on **keys** of **other**. **Join** uses **pd.merge** under the covers.

Cleaning Data with Missing Values

Pandas represents missing values as **NaN** (Not a Number). It comes from Numpy and is of type **float64**. Pandas has many methods to find and replace missing values.

Find Missing Values

```
> s_df.isnull()     or    > pd.isnull(obj)
> s_df.notnull()    or    > pd.notnull(obj)
```

Replacing Missing Values

`s_df.loc[s_df.isnull()] = 0`	Use mask to replace **NaN**
`s_df.interpolate(method='linear')`	Interpolate using different methods
`s_df.fillna(method='ffill')`	Fill forward (last valid value)
`s_df.fillna(method='bfill')`	Or backward (next valid value)
`s_df.dropna(how='any')`	Drop rows if any value is **NaN**
`s_df.dropna(how='all')`	Drop rows if all values are **NaN**
`s_df.dropna(how='all', axis=1)`	Drop across columns instead of rows

ENTHOUGHT

Take your Pandas skills to the next level! Register at www.enthought.com/pandas-master-class

Reshaping Dataframes and Pivot Tables

Tools for reshaping **DataFrames** from the *wide* to the *long* format and back. The *long* format can be *tidy*, which means that "each variable is a column, each observation is a row"[1]. Tidy data is easier to filter, aggregate, transform, sort, and pivot. Reshaping operations often produce multi-level indices or columns, which can be sliced and indexed.

1 Hadley Wickham (2014) "Tidy Data", http://dx.doi.org/10.18637/jss.v059.i10

MultiIndex: A Multi-Level Hierarchical Index

Often created as a result of:

```
> df.groupby(list_of_columns)
> df.set_index(list_of_columns)
```

Contiguous labels are *displayed* together but apply to each row. The concept is similar to multi-level columns.

A **MultiIndex** allows indexing and slicing one or multiple levels at once. Using the *Long* example from the right:

```
long.loc[1900]                    All 1900 rows
long.loc[(1900, 'March')]         value 2
long.xs('March', level='Month')   All March rows
```

Simpler than using boolean indexing, for example:

```
> long[long.Month == 'March']
```

Long to Wide Format and Back with **stack()** and **unstack()**

Pivot **column** level **to index**, i.e. "stacking the columns" (wide to long):

```
> df.stack()
```

Pivot **index** level **to columns**, "unstack the columns" (long to wide):

```
> df.unstack()
```

If multiple indices or column levels, use level number or name to **stack**/**unstack**:

```
> df.unstack(0)   or   > df.unstack('Year')
```

A common use case for unstacking, plotting group data vs index after groupby:

```
> (df.groupby(['A', 'B])['relevant'].mean()
     .unstack().plot())
```

Wide

Year	Jan.	Feb.	Mar.
1900	1	7	2
2000	4	3	9

Stack →
← Unstack

Long

Year	Month	Value
1900	Jan.	1
	Feb	7
	Mar.	2
2000	Jan.	4
	Feb	3
	Mar.	9

Pivot Tables

```
> pd.pivot_table(df,
      index=cols,  (keys to group by for index)
      columns=cols2,  (keys to group by for columns)
      values=cols3,  (columns to aggregate)
      aggfunc='mean')  (what to do with repeated values)
```

Omitting index, columns, or values will use all remaining columns of df. You can "pivot" a table manually using **groupby**, **stack** and **unstack**.

Index

Columns

0	Recently updated	Number of stations	Continent code
1	FALSE	1	EU
2	FALSE	1	EU
3	FALSE	1	EU
4	TRUE	1	EU
5	FALSE	1	AN
6	TRUE	1	AN
7	TRUE	1	AN

Continent code	AN	EU
Recently updated		
FALSE	1	3
TRUE	2	1

```
pd.pivot_table(df,
      index="Recently updated",
      columns="continent code",
      values="Number of Stations",
      aggfunc=np.sum)
```

From Wide to Long with **melt**

Specify which columns are identifiers (**id_vars**, values will be repeated for each row) and which are "measured variables" (**value_vars**, will become values in *variable* column. All remaining columns by default).

```
pd.melt(df, id_vars=id_cols, value_vars=value_columns)
```

```
pd.melt(team, id_vars=['Color'],
        value_vars=['A', 'B', 'C'],
        var_name='Team', value_name='Score')
```

Team

	Color	A	B	C
0	Red	1	3	4
1	Blue	2	-	6

Melt →

	Color	Team	Score
0	Red	A	1
1	Blue	A	2
2	Red	B	3
3	Blue	B	-
4	Red	C	4
5	Blue	C	5

df.pivot() vs **pd.pivot_table**

df.pivot() Does not deal with repeated values in index. It's a declarative form of **stack** and **unstack**.

pd.pivot_table() Use if you have repeated values in index (specify **aggfunc** argument).

Red Panda
Ailurus fulgens

ENTHOUGHT

Take your Pandas skills to the next level! Register at www.enthought.com/pandas-master-class

Getting Started with doParallel and foreach

Steve Weston*and Rich Calaway
doc@revolutionanalytics.com

September 19, 2017

1 Introduction

The `doParallel` package is a "parallel backend" for the `foreach` package. It provides a mechanism needed to execute `foreach` loops in parallel. The `foreach` package must be used in conjunction with a package such as `doParallel` in order to execute code in parallel. The user must register a parallel backend to use, otherwise `foreach` will execute tasks sequentially, even when the %dopar% operator is used.[1]

The `doParallel` package acts as an interface between `foreach` and the `parallel` package of R 2.14.0 and later. The `parallel` package is essentially a merger of the `multicore` package, which was written by Simon Urbanek, and the `snow` package, which was written by Luke Tierney and others. The `multicore` functionality supports multiple workers only on those operating systems that support the `fork` system call; this excludes Windows. By default, `doParallel` uses `multicore` functionality on Unix-like systems and `snow` functionality on Windows. Note that the `multicore` functionality only runs tasks on a single computer, not a cluster of computers. However, you can use the `snow` functionality to execute on a cluster, using Unix-like operating systems, Windows, or even a combination. It is pointless to use `doParallel` and `parallel` on a machine with only one processor with a single core. To get a speed improvement, it must run on a machine with multiple processors, multiple cores, or both.

*Steve Weston wrote the original version of this vignette for the doMC package. Rich Calaway adapted the vignette for doParallel.

[1] `foreach` will issue a warning that it is running sequentially if no parallel backend has been registered. It will only issue this warning once, however.

2 A word of caution

Because the `parallel` package in `multicore` mode starts its workers using `fork` without doing a subsequent `exec`, it has some limitations. Some operations cannot be performed properly by forked processes. For example, connection objects very likely won't work. In some cases, this could cause an object to become corrupted, and the R session to crash.

3 Registering the `doParallel` parallel backend

To register `doParallel` to be used with `foreach`, you must call the `registerDoParallel` function. If you call this with no arguments, on Windows you will get three workers and on Unix-like systems you will get a number of workers equal to approximately half the number of cores on your system. You can also specify a cluster (as created by the `makeCluster` function) or a number of cores. The `cores` argument specifies the number of worker processes that `doParallel` will use to execute tasks, which will by default be equal to one-half the total number of cores on the machine. You don't need to specify a value for it, however. By default, `doParallel` will use the value of the "cores" option, as specified with the standard "options" function. If that isn't set, then `doParallel` will try to detect the number of cores, and use one-half that many workers.

Remember: unless `registerDoMC` is called, `foreach` will *not* run in parallel. Simply loading the `doParallel` package is not enough.

4 An example `doParallel` session

Before we go any further, let's load `doParallel`, register it, and use it with `foreach`. We will use `snow`-like functionality in this vignette, so we start by loading the package and starting a cluster:

```
> library(doParallel)
> cl <- makeCluster(2)
> registerDoParallel(cl)
> foreach(i=1:3) %dopar% sqrt(i)

[[1]]
[1] 1

[[2]]
[1] 1.414214
```

```
[[3]]
[1] 1.732051
```

To use `multicore`-like functionality, we would specify the number of cores to use instead (but note that on Windows, attempting to use more than one core with `parallel` results in an error):

```
library(doParallel}
registerDoParallel(cores=2)
foreach(i=1:3) %dopar% sqrt(i)
```

> Note well that this is *not* a practical use of `doParallel`. This is our "Hello, world" program for parallel computing. It tests that everything is installed and set up properly, but don't expect it to run faster than a sequential `for` loop, because it won't! `sqrt` executes far too quickly to be worth executing in parallel, even with a large number of iterations. With small tasks, the overhead of scheduling the task and returning the result can be greater than the time to execute the task itself, resulting in poor performance. In addition, this example doesn't make use of the vector capabilities of `sqrt`, which it must to get decent performance. This is just a test and a pedagogical example, *not* a benchmark.

But returning to the point of this example, you can see that it is very simple to load `doParallel` with all of its dependencies (`foreach`, `iterators`, `parallel`, etc), and to register it. For the rest of the R session, whenever you execute `foreach` with `%dopar%`, the tasks will be executed using `doParallel` and `parallel`. Note that you can register a different parallel backend later, or deregister `doParallel` by registering the sequential backend by calling the `registerDoSEQ` function.

5 A more serious example

Now that we've gotten our feet wet, let's do something a bit less trivial. One good example is bootstrapping. Let's see how long it takes to run 10,000 bootstrap iterations in parallel on 2 cores:

```
> x <- iris[which(iris[,5] != "setosa"), c(1,5)]
> trials <- 10000
> ptime <- system.time({
+   r <- foreach(icount(trials), .combine=cbind) %dopar% {
+     ind <- sample(100, 100, replace=TRUE)
+     result1 <- glm(x[ind,2]~x[ind,1], family=binomial(logit))
+     coefficients(result1)
+   }
+ })[3]
> ptime
```

```
elapsed
 19.219
```

Using `doParallel` and `parallel` we were able to perform 10,000 bootstrap iterations in 19.219 seconds on 2 cores. By changing the `%dopar%` to `%do%`, we can run the same code sequentially to determine the performance improvement:

```
> stime <- system.time({
+   r <- foreach(icount(trials), .combine=cbind) %do% {
+     ind <- sample(100, 100, replace=TRUE)
+     result1 <- glm(x[ind,2]~x[ind,1], family=binomial(logit))
+     coefficients(result1)
+   }
+ })[3]
> stime
```

```
elapsed
 33.393
```

The sequential version ran in 33.393 seconds, which means the speed up is about 1.7 on 2 workers.[2] Ideally, the speed up would be 2, but no multicore CPUs are ideal, and neither are the operating systems and software that run on them.

At any rate, this is a more realistic example that is worth executing in parallel. We do not explain what it's doing or how it works here. We just want to give you something more substantial than the `sqrt` example in case you want to run some benchmarks yourself. You can also run this example on a cluster by simply reregistering with a cluster object that specifies the nodes to use. (See the `makeCluster` help file for more details.)

6 Getting information about the parallel backend

To find out how many workers `foreach` is going to use, you can use the `getDoParWorkers` function:

```
> getDoParWorkers()
```

```
[1] 2
```

[2] If you build this vignette yourself, you can see how well this problem runs on your hardware. None of the times are hardcoded in this document. You can also run the same example which is in the examples directory of the `doParallel` distribution.

This is a useful sanity check that you're actually running in parallel. If you haven't registered a parallel backend, or if your machine only has one core, `getDoParWorkers` will return one. In either case, don't expect a speed improvement. `foreach` is clever, but it isn't magic.

The `getDoParWorkers` function is also useful when you want the number of tasks to be equal to the number of workers. You may want to pass this value to an iterator constructor, for example.

You can also get the name and version of the currently registered backend:

```
> getDoParName()

[1] "doParallelSNOW"

> getDoParVersion()

[1] "1.0.11"
```

This is mostly useful for documentation purposes, or for checking that you have the most recent version of `doParallel`.

7 Specifying multicore options

When using `multicore`-like functionality, the `doParallel` package allows you to specify various options when running `foreach` that are supported by the underlying `mclapply` function: "preschedule", "set.seed", "silent", and "cores". You can learn about these options from the `mclapply` man page. They are set using the `foreach .options.multicore` argument. Here's an example of how to do that:

```
mcoptions <- list(preschedule=FALSE, set.seed=FALSE)
foreach(i=1:3, .options.multicore=mcoptions) %dopar% sqrt(i)
```

The "cores" options allows you to temporarily override the number of workers to use for a single `foreach` operation. This is more convenient than having to re-register `doParallel`. Although if no value of "cores" was specified when `doParallel` was registered, you can also change this value dynamically using the `options` function:

```
options(cores=2)
getDoParWorkers()
options(cores=3)
getDoParWorkers()
```

If you did specify the number of cores when registering `doParallel`, the "cores" option is ignored:

```
registerDoParallel(4)
options(cores=2)
getDoParWorkers()
```

As you can see, there are a number of options for controlling the number of workers to use with `parallel`, but the default behaviour usually does what you want.

8 Stopping your cluster

If you are using `snow`-like functionality, you will want to stop your cluster when you are done using it. The `doParallel` package's `.onUnload` function will do this automatically if the cluster was created automatically by `registerDoParallel`, but if you created the cluster manually you should stop it using the `stopCluster` function:

```
stopCluster(cl)
```

9 Conclusion

The `doParallel` and `parallel` packages provide a nice, efficient parallel programming platform for multiprocessor/multicore computers running operating systems such as Linux and Mac OS X. It is very easy to install, and very easy to use. In short order, an average R programmer can start executing parallel programs, without any previous experience in parallel computing.

SQL Facts

- SQL stands for Structured Query Language
- SQL is pronounced "sequel"
- SQL is declarative language
- SQL is used to access & manipulate data in databases
- Top SQL DBs are MS SQL Server, Oracle, DB2, and MySQL

Database Definitions

- **RDBMS** (Relational Database Management System) – Software that stores and manipulates data arranged in relational database tables.
- **Table** – A set of data arranged in columns and rows. The columns represent characteristics of stored data and the rows represent actual data entries.

SQL Commands Categories

Data Query Language (DQL)

- SELECT - Retrieve data from table(s)

Data Manipulation Language (DML)

- INSERT - Insert data into db table
- UPDATE - Update data in db table
- DELETE - Delete data from table

Data Definition Language (DDL)

- CREATE - Create db object (table, view, etc.)
- ALTER - Modify db object (table, view, etc.)
- DROP - Delete db object (table, view, etc.)

Data Control Language (DCL)

- GRANT - Assign privilege
- REVOKE - remove privilege

How to select data from a table

```
SELECT <Column List>
FROM <Table Name>
WHERE <Search Condition>
```

Example:

```
SELECT FirstName, LastName, OrderDate
FROM Orders WHERE OrderDate > '10/10/2010'
```

How to insert data in a table

```
INSERT INTO <Table Name>
(<Column List>) VALUES (<Values>)
```

Example:

```
INSERT INTO Orders
(FirstName, LastName, OrderDate) VALUES
('John', 'Smith', '10/10/2010')
```

How to update data in a table

```
UPDATE <Table Name>
SET <Column1> = <Value1>, <Column2> = <Value2>, …
WHERE <Search Condition>
```

Example:

```
UPDATE Orders
SET FirstName = 'John', LastName = 'Who' WHERE LastName='Wo'
```

How to delete data from a table

```
DELETE FROM <Table Name>
WHERE <Search Condition>
```

Example:

```
DELETE FROM Orders
WHERE OrderDate < '10/10/2010'
```

How to group data and use aggregates

```
SELECT <Column List>, <Aggregate Function>(<Column Name>)
FROM <Table Name>
WHERE <Search Condition>
GROUP BY <Column List>
```

Example:

```
SELECT LastName, SUM(OrderValue)
FROM Orders
WHERE OrderDate > '10/10/2010'
GROUP BY LastName
```

How to order data

```
SELECT <Column List>
FROM <Table Name>
WHERE <Search Condition>
ORDER BY <Column List>
```

Example:

```
SELECT FirstName, LastName, OrderDate
FROM Orders
WHERE OrderDate > '10/10/2010'
ORDER BY OrderDate
```

How to select data from more than one table

```
SELECT <Column List>
FROM <Table1> JOIN <Table2>
ON <Table1>.<Column1> = <Table2>.<Column1>
```

Example:

```
SELECT Orders.LastName, Countries.CountryName
FROM Orders JOIN Countries ON
Orders.CountryID = Countries.ID
```

Using UNION

```
SELECT <Column List> FROM <Table1>
UNION
SELECT <Column List> FROM <Table2>
```

Example:

```
SELECT FirstName, LastName FROM Orders2010
UNION
SELECT FirstName, LastName FROM Orders2011
```

CREATE TABLE

```
CREATE TABLE <Table Name>
(   Column1 DataType,
    Column2 DataType,
    Column3 DataType,
    ....    )
```

```
CREATE TABLE Orders
(   FirstName CHAR(100),
    LastName CHAR(100),
    OrderDate DATE,
    OrderValue Currency    )
```

TensorFlow Cheat Sheet

About

TensorFlow

TensorFlow™ is an open source software library for numerical computation using data flow graphs. TensorFlow was originally developed for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well.

Skflow

Scikit Flow provides a set of high level model classes that you can use to easily integrate with your existing Scikit-learn pipeline code. Scikit Flow is a simplified interface for TensorFlow, to get people started on predictive analytics and data mining. Scikit Flow has been merged into TensorFlow since version 0.8 and now called TensorFlow Learn.

Keras

Keras is a minimalist, highly modular neural networks library, written in Python and capable of running on top of either TensorFlow or Theano

Installation

How to install new package in Python:

`pip install <package-name>`

Example: `pip install requests`

How to install tensorflow?

device = **cpu/gpu**

python_version = **cp27/cp34**

```
sudo pip install
https://storage.googleapis.com/
tensorflow/linux/$device/tensorflow-
0.8.0-$python_version-none-linux_x86
_64.whl
```

How to install Skflow

`pip install sklearn`

How to install Keras

`pip install keras`

update ~/.keras/keras.json - replace "theano" by "tensorflow"

Helpers

Python helper

Important functions

`type(object)`

Get object type

`help(object)`

Get help for object (list of available methods, attributes, signatures and so on)

`dir(object)`

Get list of object attributes (fields, functions)

`str(object)`

Transform an object to string

`object?`

Shows documentations about the object

`globals()`

Return the dictionary containing the current scope's global variables.

`locals()`

Update and return a dictionary containing the current scope's local variables.

`id(object)`

Return the identity of an object. This is guaranteed to be unique among simultaneously existing objects.

```
import __builtin__
dir(__builtin__)
```

Other built-in functions

TensorFlow

Main classes

```
tf.Graph()
tf.Operation()
tf.Tensor()
tf.Session()
```

Some useful functions

```
tf.get_default_session()
tf.get_default_graph()
tf.reset_default_graph()
ops.reset_default_graph()
tf.device("/cpu:0")
tf.name_scope(value)
tf.convert_to_tensor(value)
```

TensorFlow Optimizers

```
GradientDescentOptimizer
AdadeltaOptimizer
AdagradOptimizer
MomentumOptimizer
AdamOptimizer
FtrlOptimizer
RMSPropOptimizer
```

Reduction

```
reduce_sum
reduce_prod
reduce_min
reduce_max
reduce_mean
reduce_all
reduce_any
accumulate_n
```

Activation functions

`tf.nn?`

relu
relu6
elu
softplus
softsign
dropout
bias_add
sigmoid
tanh
sigmoid_cross_entropy_with_logits
softmax
log_softmax
softmax_cross_entropy_with_logits
sparse_softmax_cross_entropy_with_logits
weighted_cross_entropy_with_logits
etc.

Skflow

Main classes

TensorFlowClassifier
TensorFlowRegressor
TensorFlowDNNClassifier
TensorFlowDNNRegressor
TensorFlowLinearClassifier
TensorFlowLinearRegressor
TensorFlowRNNClassifier
TensorFlowRNNRegressor
TensorFlowEstimator

Each classifier and regressor have following fields

n_classes=0 (Regressor), n_classes are expected to be input (Classifiers)

batch_size=32,
steps=200, // except TensorFlowRNNClassifier - there is 50
optimizer='Adagrad',
learning_rate=0.1,
class_weight=None,
clip_gradients=5.0,
continue_training=False,
config=None,
verbose=1.

Each class has a method fit

`fit(X, y, monitor=None, logdir=None)`

X: matrix or tensor of shape [n_samples, n_features...]. Can be iterator that returns arrays of features. The training input samples for fitting the model.

Y: vector or matrix [n_samples] or [n_samples, n_outputs]. Can be iterator that returns array of targets. The training target values (class labels in classification, real numbers in regression).

monitor: Monitor object to print training progress and invoke early stopping

logdir: the directory to save the log file that can be used for optional visualization.

`predict (X, axis=1, batch_size=None)`

Args:

X: array-like matrix, [n_samples, n_features...] or iterator.

axis: Which axis to argmax for classification.

By default axis 1 (next after batch) is used. Use 2 for sequence predictions.

batch_size: If test set is too big, use batch size to split it into mini batches. By default the batch_size member variable is used.

Returns:

y: array of shape [n_samples]. The predicted classes or predicted value.

Useful links

TensorFlow API
https://www.tensorflow.org/versions/r0.9/api_docs/index.html
TensorFlow How-Tos
https://www.tensorflow.org/versions/r0.9/how_tos/index.html
Skflow github
https://github.com/tensorflow/skflow
Skflow API
http://scikit-learn.org/stable/modules/classes.html#module-sklearn.cluster
Keras
http://keras.io/
Our github
https://github.com/Altoros/TensorFlow-training
TensorFlow github
https://github.com/tensorflow/tensorflow