Chapter 1 : Python Fundamentals

Our assumption of starting with this module is that you have already installed Anaconda distribution for python using links given on LMS, although a better option is to simply Google `Download Anaconda` and follow the links [As link on LMS might be an old one , not updated as per the latest release]. Key things before we start with python programming

1. Make sure you go through the videos and have chosen a python editor which you like . (We use Jupyter notebooks in the course , scripts provided will be notebooks. You can use spyder as well, there will be no difference in syntax.)
2. Make sure that you code along , with videos as well as this book. There is no shortcut to learn programming except writing/modifying code on your own, executing and trouble shooting.

Lets begin

Creating Basic Objects with single values

Choose any name and equate to the value which you wish the object to contain. There are few restrictions however on choosing object names which we will discuss few steps later

```
x=4
y=5
x+y
```

```
9
```

For checking what value a particular object holds , you can simply write the object name in the cell and execute ; object value will be displayed in the output .

```
x
```

```
4
```

Just like R, everything in python is also case sensitive , if we now try to execute `x` [X in caps]

```
X
```

```
NameError                 Traceback (most recent call last)

 in
----> 1 X
NameError: name 'X' is not defined
```

You can see that python doesn't recognize X in caps as an object name because we have not created one. We named our object a lowercase x . As an important side note , Error messages in python are printed with complete traceback [Most of which is generally not very helpful] , best place to start debugging is from the bottom.

Few basic rules to follow when choosing object names :

1. Name should not start with a number
2. Name should not have any space in between
3. No special character is allowed except underscore `_`
4. These reserved key words should not be used as object names

```
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

Object Types and Typecasting

Although we don't need to declare type of objects at the time of creation explicitly , python does assign type to variables on its own. Anything between quotes (there is no difference between single or double quotes) is considered to be of string/character type.

```
x=25
y='lalit'
type(x)
```

```
int
```

```
type(y)
```

```
str
```

`int` here means integer and `str` means string. This assigned type is important as operations are allowed/not allowed depending on type of variables , irrespective of what values are stored in them .

This type assignment to objects affects/dictates what kind of operations are allowed on/with the object. For example adding a number to a string is going to throw and error. Lets try doing that anyway and see what happens .

```
x='34' # python will consider this to be a string because the value is within
quotes
x+3
```

```
TypeError                                 Traceback (most recent call last)
 in
1 x='34' # python will consider this to be a string because the value is within quotes
----> 2 x+3
TypeError: must be str, not int
```

However we can change the type of object using typecasting function and then do the operations .

```
int(x)+3
```

```
37
```

Just applying this function on x , doesn't change its type permanently , it outputs an integer on which the addition operation could be done . Type of x is still `str`

```
type(x)
```

```
str
```

If we do want to change the type of object , we'd need to equate the output of typecasting function to the original object itself

```
x=int(x)
type(x)
```

```
int
```

Last thing about typecasting , unlike R , if the data contained in a object is such that, it can not be converted to a certain type ; then you'll get an error instead of `missing value` as output

```
int('king')
```

```
ValueError                                Traceback (most recent call last)
```

```
 in
----> 1 int('king')
ValueError: invalid literal for int() with base 10: 'king'
```

Numeric Operations

Usual Arithmetic operations can be used for numeric type objects (int , float , bool) by using appropriate universal symbols, lets look at some examples

```
x=2
y=19
x+y
```

```
21
```

```
x-y
```

```
-17
```

```
x*y
```

```
38
```

```
x/y
```

```
0.10526315789473684
```

For exponents/power , python exclusively uses double asterisks . `^` operator which can be used for exponents in R , should not be used in python for exponents (it does bit-wise OR operation)

```
x**y
```

```
524288
```

You can of-course right complex calculations using parenthesis . In order to save the results of these computations , you need to simply equate them to an object name. Notice , when you do that, an explicit output will not be printed as we saw earlier

```
z=(x/y)**(x*y+3)
```

If you want to see the outcome , you can simply type the object name where you stored to result

```
z
```

```
8.190910549494282e-41
```

for using mathematical function you'll need to use package math

```
import math
math.log(34)
```

```
3.5263605246161616
```

```
math.exp(3.5263605246161616)
```

```
34.00000000000001
```

Boolean Objects

There are two values which Boolean objects can take `True` and `False`. They need to be spelled as is for them to be considered Boolean values

```
x=True
y=False
type(x),type(y)
```

(bool, bool)

you can do usual Boolean operations using both words and symbols

```
x and y , x & y
```

(False, False)

```
x or y , x | y
```

(True, True)

```
not x , not y
```

(False, True)

Note that for negation , `!` is not being used as we did in R. In python you simply use the keyword `not` for reversing Boolean values

Writing Conditions

In practice , it doesn't happen too often that we create objects with Boolean values explicitly . Boolean values are usually results of conditions that we apply on other objects containing data. Here are some examples

Equality Condition

```
x=34
y=12
x==y
```

False

Since x is not equal to y, outcome of the condition is `False`. Note carefully that for writing equality condition , we used two equal to sign , one simple equal to sign is reserved for assignment

In-Equality Condition

```
x!=y
```

True

Greater than , Less than

```
x>y
```

True

```
x<y
```

False

Greater than or equal to , Less than or equal to

```
x>= y
```

True

```
x<=y
```

> False

These conditions can be written for character data also . in case of equality , strings should match exactly including lower/upper case of characters as well. In case of less than , greater than kind of comparison , result [True/False] depends on dictionary order of the strings [Not their length]

Membership condition for iterables

you can use operators `in` and `not in` to check if some string/element is present in a string/list . The examples here do not show this in context of lists , because we haven't yet introduced them

```
x='python'
'py' in x
```

> True

```
'TH' in x # all string comparisons are case sensitive
```

> False

Compound Conditions

you can use parenthesis to write a compound conditions which is essentially is combination of multiple individual conditions here is an example

```
x=45
y=67
(x > 20 and y<10) or (x ==4 or y > 15 )
```

> True

Notice that , eventual result of the condition is a single Boolean value

String Operations

```
x='Python'
y="Data"
```

All string data will need to be within quotes . Note that it doesn't matter whether you are using double quotes or single quotes

length of a string (iterable)

```
len(x),len(y)
```

> (6, 4)

same function `len` will also work in context of lists

Duplicating strings

When a string is multiplied by an integer (not decimals/float) , it results in duplication of the base string

```
'python'*3
```

> 'pythonpythonpython'

Concatenating strings

Addition operator `+` when used between strings does concatenation

```
x+y
```

'PythonData'

```
x+' and '+y
```

'Python and Data'

```
z=x+' and '+y
z
```

'Python and Data'

Converting to lower case

```
z.lower()
```

'python and data'

```
z
```

'Python and Data'

Note that these functions are giving explicit output [not making inplace changes in the object itself]. If you want to change z itself , you'll need to equate the function call to object it self

```
z=z.lower()
z
```

'python and data'

Converting to upper case

```
z.upper()
```

'PYTHON AND DATA'

Proper case [first letter capital]

```
z.capitalize()
```

'Python and data'

Adding white spaces to string

```
z.rjust(20)
```

' python and data'

we just added some white spaces [to left side] to our string z, note that the number passed to function `rjust` represents length of the string after adding white spaces. It does not represent number of white spaces being added to the string. If this input is smaller than the string in question , then the result is same as the original string with not changes whatsoever

```
len(z.rjust(20))
```

20

```
z.rjust(3)
```

'python and data'

try for yourself and see what these functions do : `ljust` , `center`

Removing white spaces from a string

lets first add leading and trailing spaces to our string and then we'll learn about how to remove those spaces using available string functions in python

```
z=z.center(30)
z
```

```
'      python and data      '
```

```
z.strip() # removes both leading and trailing white spaces
```

```
'python and data'
```

```
z.rstrip() # removes only trailing white spaces
```

```
'      python and data'
```

```
z.lstrip() # removes only leading white spaces
```

```
'python and data      '
```

Replacing a substring with another

```
z.replace('a',"@#!")
```

```
'      python @#!nd d@#!t@#!      '
```

- first argument represents substring to be replaced
- second argument represents the substring which will replace the substring mentioned in first argument
- By default , all occurrences are replaced , however you can use 3rd argument to the function to control that , here is an example

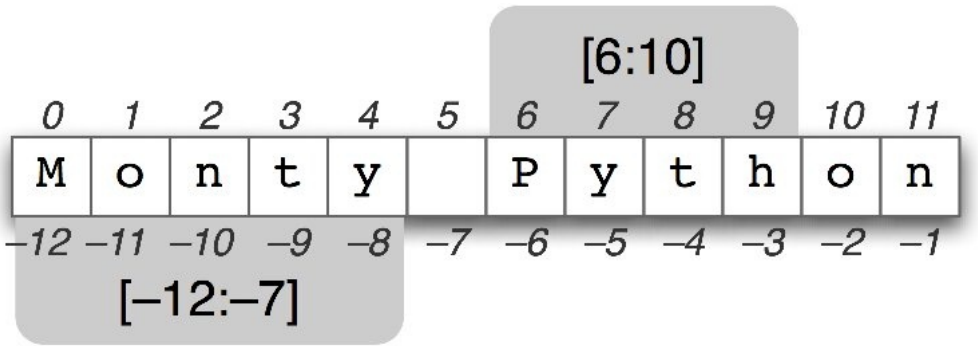
```
z.replace('a',"@#!",2)
```

```
'      python @#!nd d@#!ta      '
```

this only replaces first 2 occurrences as specified in the third argument

Position indices and access for iterables [Strings , Lists]

```
x='Monty Python'
```



In Figure above you can see that each individual character of the string (including white spaces) is assigned an index. From left to right index starts with 0 . From right to left , index starts with -1.

You can use these indices to extract parts of substring.

```
x[6]
```

'P'

```
x[-9]
```

't'

multiple characters can be extracted as well [contiguous ranges only] by passing range of indices

```
x[6:10]
```

'Pyth'

```
x[-12:-7]
```

'Monty'

Note that x[a:b] will give you part of string starting with index `a` and ending with index `b-1` [last value is not included in the output]. By default this assumes step size 1; from left to right. If starting position happens to be occurring after the ending position , result is simply an empty string [instead of an error]

```
x[-7:-12]
```

''

You can change step size by making use of third input here

```
x[1:9]
```

'onty Pyt'

```
x[1:9:2]
```

'ot y'

step size 2 means that , starting from beginning , the next element will be at step size 2

you can also have step size negative , in that case , direction changes to :: right to left

```
x[-7:-12:-1]
```

' ytno'

Just like step size (which we don't always specify), we don't need to specify starting and ending points either. In absence of starting position , first value of the strings becomes the default starting point . In absence of ending position, last value of the strings becomes the default end point.

```
x[:4]
```

'Mont'

```
x[5:]
```

' Python'

Note : Same way of indexing will be used for lists as well as mentioned in the title. However in case of lists , index will assigned to each individual element. Rest of the behavior will be same as strings , as we discussed here

Lists

Lists are our first encounter with data structures in python. Objects which can hold more than one values. Many other data structures that we are going to come across in python are derived from these.

Lists are collection of multiple objects (of any kind), you can create them by simply putting the values within square brackets `[]` separated by commas

```
x=[20,92,43,83,"john","a","c",45]
len(x)
```

```
8
```

In general you are going to find out, that when you apply len function on a data structure object , outcome is going to be how many elements they contain.

Indexing in context of lists works just like strings as mentioned earlier.Only difference being that, **for lists each individual element is assigned an index.** Here are some examples of using indices with lists. Notice the similarity with strings

```
x[4]
```

```
'john'
```

```
x[-2]
```

```
'c'
```

```
x[:4]
```

```
[20, 92, 43, 83]
```

```
x[2:]
```

```
[43, 83, 'john', 'a', 'c', 45]
```

```
x[1:9:2]
```

```
[92, 83, 'a', 45]
```

```
x[-12:-5]
```

```
[20, 92, 43]
```

```
x[-5:-12]
```

```
[]
```

```
x[-5:-12:-1]
```

```
[83, 43, 92, 20]
```

Reassigning existing values in a list

The way we access elements of lists , we can re-assign them as well.

```
x
```

```
[20, 92, 43, 83, 'john', 'a', 'c', 45]
```

```
x[3]
```

```
83
```

```
x[3]='python'
x
```

[20, 92, 43, 'python', 'john', 'a', 'c', 45]

Its not necessary that assigned value has the same type as the original one . We can re-assign multiple values also, keeping in mind that there, the reassignment should be done with list of values [its not necessary that number of values should be as many as the original ones]

```
x
```

[20, 92, 43, 'python', 'john', 'a', 'c', 45]

```
x[2:6]
```

[43, 'python', 'john', 'a']

```
x[2:6]=[-10,'doe',-20]
x
```

[20, 92, -10, 'doe', -20, 'c', 45]

Adding new values to a list

There are 4 ways in which we can add new elements to a list

- Simply sum two lists
- Use function append , it adds elements to the end of the list. if you try to append a list to a list, entire list gets added as a single element [we'll see examples]
- Use function extend , this also adds elements to the end of the list. if you try to use extend to add a list to a list, elements become individual elements of the new list [we'll see examples]
- Use function insert to insert values at a specific position in the list

```
x=[20,92,43,83]
x=x+[-10,12]
x
```

[20, 92, 43, 83, -10, 12]

```
x.append(100) # note that, this makes in-place changes to x
x
```

[20, 92, 43, 83, -10, 12, 100]

Notice what happens when try to add list to a list using append

```
len(x)
```

7

```
x.append(['a','b','c'])
x
```

[20, 92, 43, 83, -10, 12, 100, ['a', 'b', 'c']]

```
len(x)
```

8

```
x[7]
```

['a', 'b', 'c']

lets try to do similar thing with function extend instead

```
x.extend([3,5,7])
x
```

> [20, 92, 43, 83, -10, 12, 100, ['a', 'b', 'c'], 3, 5, 7]

```
len(x)
```

> 11

this time all elements of the list , get added as individual elements

first 3 methods here , however don't enable us to add elements in any desired position in the list. For that we need to use function `insert`

```
x
```

> [20, 92, 43, 83, -10, 12, 100, ['a', 'b', 'c'], 3, 5, 7]

```
x.insert(4,'python')
x
```

> [20, 92, 43, 83, 'python', -10, 12, 100, ['a', 'b', 'c'], 3, 5, 7]

There is now an additional element in the list at index 4

Removing elements from a list

function `pop` removes values from specified location. If any location is not specified , it simply removes the last element from the list

```
x
```

> [20, 92, 43, 83, 'python', -10, 12, 100, ['a', 'b', 'c'], 3, 5, 7]

```
x.pop()
```

> 7

```
x
```

> [20, 92, 43, 83, 'python', -10, 12, 100, ['a', 'b', 'c'], 3, 5]

If you specify a position, as shown in the next example, element in that index gets removed from the list

```
x.pop(4)
```

> 'python'

```
x
```

> [20, 92, 43, 83, -10, 12, 100, ['a', 'b', 'c'], 3, 5]

This however might be a big hassle if we don't have prior information on where in the list value resides which we want to remove . function `remove` comes to your rescue .

```
x.remove(83)
x
```

> [20, 92, 43, -10, 12, 100, ['a', 'b', 'c'], 3, 5]

This will throw an error if the value doesn't exist in the list

```
x.remove(83)
```

```
ValueError                                Traceback (most recent call last)

 in
----> 1 x.remove(83)
ValueError: list.remove(x): x not in list
```

Another thing to note is that, if there are multiple occurrences of the same value in the list then at a time only first occurrence will be removed. You might need to run remove function in a loop (which we will learn about in some time)

Rearranging values of a list

```
x=[2,3,40,2,14,2,3,11,71,26]
x.sort()
x
```

```
[2, 2, 2, 3, 3, 11, 14, 26, 40, 71]
```

default order of sorting is ascending , we can use option `reverse` and set it to `True` if sorting in descending manner is required

```
x=[2,3,40,2,14,2,3,11,71,26]
x.sort(reverse=True)
x
```

```
[71, 40, 26, 14, 11, 3, 3, 2, 2, 2]
```

Many at times we might need to simply flip [reverse order] the values of a list without really doing any sorting. function `reverse` can be used for that .

```
x=[2,3,40,2,14,2,3,11,71,26]
x.reverse()
x
```

```
[26, 71, 11, 3, 2, 14, 2, 40, 3, 2]
```

Flow control in python

so far we have been looking at object creation and then modifying them in various ways and doing operations on them. We as such haven't done anything which follows couple of logical decisions in the process. In this section we'll be learning few new tools in python which enables to include decision making in the programming.

We'll also be learning to write repetitive codes in a more concise manner with for and while loops.

we'll start with if-else code blocks

if-else

```
x=12
if x%2==0:
    print(x,' is even')
else :
    print(x,'x is odd')
```

```
12  is even
```

couple of important things to learn here , both in context of if-else block and in general about python.

- a condition follows after the keyword if
- colon indicates , end of condition and start of the code block if
- if the condition is true , program written inside the `if` block will be executed
- if the condition is not true , program written in `else` block will be executed
- Notice the **indentation** , instead of curly braces to define code blocks, in python levels of indentations are used , this makes the code easy to read

Lets look at one more example to understand functionality of if-else block and importance of indentation. Lets say , given 3 numbers we are trying to find maximum value among them

```
a,b,c=30,10,-10 # you can assign values at once like this
# this doesnt work : a=3,b=10,c=-10
```

```
if a>b :
    if a>c:
        mymax=a
    else :
        mymax=c
else :
    if b>c:
        mymax=b
    else:
        mymax=c
```

```
mymax
```

```
30
```

You can experiment with passing different numbers. Couple of lessons to learn from this

- you can have code blocks inside code blocks
- level of indentation increases as we add more code blocks inside already existing one

```
x=[5,40,12,-10,0,32,4,3,6,72]
```

For loop

Lets say i wanted to do odd/even exercise for all the numbers in this list. Technically we can do this , by value of x in the code that we had written 10 times. or writing 10 if-else blocks. turns out, given the for loop, we don't need to do that . lets see

```
for element in x:
    if element%2==0:
        print(element,' is even')
    else:
        print(element,' is odd')
```

```
5  is odd
40  is even
12  is even
-10  is even
0  is even
32  is even
4  is even
3  is odd
6  is even
72  is even
```

- `element` here is known as index . it can be given any name like generic objects in python , `element` is not a fixed name

- x/value_list can be any list or in general iterable
- body of the for loop is executed as many times as there are values in the value_list

here is another example

```
cities=['Mumbai','London','Bangalore','Pune','Hyderabad']
for i in  cities:
    num_chars=len(i)
    print(i+ ':'+ str(num_chars))
```

> Mumbai:6
> London:6
> Bangalore:9
> Pune:4
> Hyderabad:9

for loops can have multiple indices as well if the value list elements themselves are lists

```
x=[[1,2],['a','b'],[34,67]]
for i,j in x:
    print(i,j)
```

> 1 2
> a b
> 34 67

how ever for multiple indices to work , all the list element within the larger list need to same number of elements

While Loop

We have seen so far that for loops work with indices iterating over a value list. Many at times , we need to do this iterative operation basis a condition instead of a value list . We can do this using a while loop . Lets see an example

Lets say we want to remove all the occurences of a value from a list. Remove function that we learned about , only removes first occurrence . We can manually keep on running call to remove until the all the occurences are removed or we can put this inside a while loop with the condition .

```
a=[3,3,4,4,43,3,3,3,2,2,45,67,89,3,3]
while 3 in a :
    a.remove(3)
```

```
a
```

> [4, 4, 43, 2, 2, 45, 67, 89]

List Comprehension

when we need to construct another list using an existing one doing some operation, usual way of doing that is to start with an empty list; iterate over the existing list, do some operation on the elements and append the result to empty list . like this :

```
x=[3,89,7,-90,10,0,9,1]
y=[]

for elem in x:
    sq=elem**2
    y.append(sq)
```

```
y
```

```
[9, 7921, 49, 8100, 100, 0, 81, 1]
```

y now contains squares of all numbers in x. There is another way of achieving this where we write the for loop (shorter version of it) inside the empty list directly

```
y = [elem**2 for elem in x]
```

```
y
```

```
[9, 7921, 49, 8100, 100, 0, 81, 1]
```

Here the operation on each elem in x is `elem**2` this becomes element of the list automatically , without having to write an explicit for loop. This is called list comprehension . We can include conditional statement also in list comprehension . Lets first look at explicit for loop for the same .

```
x= [ 4,-5,67,98,11,-20,7]
import math
y=[]
for elem in x:
    if elem>0:
        y.append(math.log(elem))

print(y)
```

```
[1.3862943611198906, 4.204692619390966, 4.584967478670572, 2.3978952727983707, 1.9459101490553132]
```

```
y=[math.log(elem) for elem in x if elem>0]
print(y)
```

```
[1.3862943611198906, 4.204692619390966, 4.584967478670572, 2.3978952727983707, 1.9459101490553132]
```

Note: usage of `print` here is just to display this horizontally in comparison to vertically the way it naturally gets displayed .

you can include else statement as well

```
logs=[math.log(num) if num>0 else 'out of domain' for num in x]
print(logs)
```

```
[1.3862943611198906, 'out of domain', 4.204692619390966, 4.584967478670572, 2.3978952727983707, 'out of domain', 1.9459101490553132]
```

One caution here , don't always push for writing a list comprehension instead of a for loop. Purpose of list comprehension is to shorten the code at the same time, not compromising on the readability of a code. They provide no performance improvement on run time. Hence if list comprehension starts to become too complex , its better to go with an explicit for loop

Dictionaries , Set and Tuples

In this section we'll learn about other data structures in python, different from lists. They are not as widely used but do come with special properties which might be useful in special cases . We'll start our discussion with dictionaries . Dictionaries , unlike lists are unordered; meaning their elements do not have any index attached to them and can not be accessed like list element by passing element index/position. Elements of a dictionary is made up of key:value pair . Here is an example

```
my_dict=
{'name':'lalit','city':'hyderabad','locality':'gachibowli','num_vehicles':2,3:78,4:[3,4,5,6]}
```

```
len(my_dict)
```

> 6

this dictionary has 6 elements (key:value pairs)

```
type(my_dict)
```

> dict

special functions associated with dictionaries let you extract keys and values of dictionaries

```
my_dict.keys()
```

> dict_keys(['name', 'city', 'locality', 'num_vehicles', 3, 4])

```
my_dict.values()
```

> dict_values(['lalit', 'hyderabad', 'gachibowli', 2, 78, [3, 4, 5, 6]])

As you can see, numbers and strings alike can be keys of dictionaries . Now that we can not really use indices to access elements of dictionaries , how do we access them ? Using keys , as follows

```
my_dict['name']
```

> 'lalit'

```
my_dict[3]
```

> 78

As you can see, output is; values associated with the keys . A dictionary does not support duplicate keys , values however have no such restriction .

For adding a new key value pair , you simply need to do this :

```
my_dict['city']='delhi'
print(my_dict)
```

> {'name': 'lalit', 'city': 'delhi', 'locality': 'gachibowli', 'num_vehicles': 2, 3: 78, 4: [3, 4, 5, 6]}

you can see that dictionary now has one more key:value pair; `'city': 'delhi'`

for removing an element , you can use keyword del

```
del my_dict[4]
print(my_dict)
```

> {'name': 'lalit', 'city': 'delhi', 'locality': 'gachibowli', 'num_vehicles': 2, 3: 78}

Although rarely used , but you can do something similar to list comprehension for dictionaries also

```
d = {elem:elem**2 for elem in x}
d
```

> {4: 16, -5: 25, 67: 4489, 98: 9604, 11: 121, -20: 400, 7: 49}

here elem has become the key of dictionary and elem**2, the associated value with it

Next we'll look at sets. Sets are like mathematical sets. They are unordered collection of unique values . You can not have duplicate elements in a set , even if you forcefully try to . They are created just like lists , but using curly braces .

```
x= {10, 2, 2, 4, 4, 4, 5, 60, 22, 76}
x
```

```
{2, 4, 5, 10, 22, 60, 76}
```

to add and remove elements , there are special functions associated with sets

```
x.add('python')
x
```

```
{10, 2, 22, 4, 5, 60, 76, 'python'}
```

```
x.remove(5)
```

As mentioned above , sets are unordered , meaning they do not have any indices associated with their elements and can not not be accessed the way we accessed lists .

```
x[2]
```

```
TypeError                                 Traceback (most recent call last)

 in
----> 1 x[2]
TypeError: 'set' object does not support indexing
```

How ever we can iterate through the elements

```
for elem in x :
    print(elem)
```

```
2
4
10
76
python
22
60
```

Notice here that order of elements printed here is not the same as it was when we created the set . This can not be predetermined either . However will remain same once created .

They also have set operation function associated with them as we find with mathematical sets.

```
a= {12,  9, 13,  6,  4, 14, 17,  5,  1,  0}
b={2, 14, 16,  4, 19, 13, 12,  6}
```

```
a.union(b)
```

```
{0, 1, 2, 4, 5, 6, 9, 12, 13, 14, 16, 17, 19}
```

```
a.intersection(b)
```

```
{4, 6, 12, 13, 14}
```

```
a.difference(b)
```

```
{0, 1, 5, 9, 17}
```

```
b.difference(a)
```

```
{2, 16, 19}
```

```
a.symmetric_difference(b)
```

> {0, 1, 2, 5, 9, 16, 17, 19}

- union : gives a new set containing elements from both a and b
- intersection : gives elements which are common in both a and b
- difference : set1.difference(set2) will output those elements of set1 which are not there in set2
- symmetric_difference : gives elements from both a and b which are **not** common between them

Sets are used in practice when the usage doesn't require iterating over or indices . They are good for maintaining collection of unique elements and checking presence of an element in sets is significantly faster in comparison to lists.

last data structure that we talk about is tuples . they are exactly same as list except one difference . You can not modify elements of tuples . They are created just like lists , except using small brackets .

```
x= ('a','b',45,98,0,-10,43)
```

tuples don't have any function like add/remove . In general there doesn't exist any way to modify a tuple once created (you can of course , change the type to list and then modify but that's beside the point)

```
x[4] = 99
```

> TypeError Traceback (most recent call last)
>
> in
> ----> 1 x[4] = 99
> TypeError: 'tuple' object does not support item assignment

Functions

A function is a block of code which only runs when it is called.You can pass data, known as parameters, into a function. A function can return data as a result (this is optional).Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable. Furthermore, it avoids repetition and makes code reusable.

Lets say we are asked to come up with a program which, when provided a list, give us a dictionary containing unique elements of the list as keys and their counts as values

```
x= [2,2,3,4,4,4,4,2,2,2,2,4,5,5,5,6,6,1,1,1,3]
count_dict={}
for elem in x:
    if elem not in count_dict:
        count_dict[elem]=1
    count_dict[elem]=count_dict[elem]+1
```

```
count_dict
```

> {2: 8, 3: 3, 4: 6, 5: 4, 6: 3, 1: 4}

Now if we need to do this multiple times in our project , we'll need to copy this entire bit of code wherever this is required. Instead of doing that we can make use of `functions` . We'll wrap this program in a function and when we need to use it, we'll have to write just one line, instead of writing the whole program.

function definitions start with keyword `def` , in the brackets which follow, we name the input which our program/function requires . In our case in discussion, there is a single input , a list . Functions can have multiple , even varying number of inputs . Lets convert , program written above into a function.

```
def my_count(a):
    count_dict={}
    for elem in a:
        if elem not in count_dict:
            count_dict[elem]=1
        count_dict[elem]+=+1
    return(count_dict)
```

now i can call this function using a single line and it will return the output back .

```
my_count(x)
```

```
{2: 8, 3: 3, 4: 6, 5: 4, 6: 3, 1: 4}
```

```
my_count([3,3,3,3,3,4,4,4,3,3,3,3,10,10,10,0,0,0,0,10,10,4,4,4])
```

```
{3: 10, 4: 7, 10: 6, 0: 5}
```

Default arguments/parameters to a function

Functions that we have been using (other than the one that we just wrote), can take variable number of inputs. If we miss passing some value they work with default value given to them . Lets see how to create a function with default values for arguments . We are going to look at a function which takes input 3 numbers and returns a weighted sum.

```
def mysum(x,y,z):

    s=100*x+10*y+z
    return(s)
```

it works fine if we pass 3 arguments while calling it

```
mysum(1,2,3)
```

```
123
```

but if we try to call it with less than 3 numbers , it starts to throw error

```
mysum(1,2)
```

```
TypeError                                 Traceback (most recent call last)

 in
----> 1 mysum(1,2)
TypeError: mysum() missing 1 required positional argument: 'z'
```

while creating the function itself, we could have given default values for it work with in order to avoid this . We can use any value as default value (only thing ensure is that , it should make sense in the context of what function does)

```
def mysum(x=1,y=10,z=-1):

    s=100*x+10*y+z
    return(s)
```

```
mysum(1,2,3)
```

```
123
```

it still works as before when we are explicitly passing all the values . However if we chose to pass lesser number of inputs , instead of throwing errors , it makes use of default values provided by us to each argument.

```
mysum(1,2)
```

```
119
```

Last thing , about the functions; when we are passing the arguments while calling the function; they are assigned to various objects in the functions in the sequence in which they are passed. This can be changed if we name our arguments while we pass them. Sequence doesn't matter in that case.

```
mysum(z=7,x=9)
```

```
1007
```

Classes

what is a class? Simply a logical grouping of data and functions . What do we mean by "logical grouping"? Well, a class can contain any data we'd like it to, and can have any functions (methods) attached to it that we please. Rather than just throwing random things together under the name "class", we try to create classes where there is a logical connection between things. Many times, classes are based on objects in the real world (like Customer or Product or Points).

Regardless, classes are a way of thinking about programs. When you think about and implement your system in this way, you're said to be performing Object-Oriented Programming. "Classes" and "objects" are words that are often used interchangeably.

Lets look at a use case which will convince you that when using a class to define a logical grouping makes your life easier

Lets say I want to keep track of customer records. Customers have lets say three attributes associated with them : name , account_balance and account_number. I will need to create three different objects for each customer and then ensure that I don't end up mixing those objects with another customer's details. I'll rather write a class for customers.

```
class customer():

    # the first function is __init__ , with double underscore
    # this is used to set attributes of object of the class customer when it gets created
    # we can also put data here which can be used by any object of the class customer
    # can also be used by other methods/functions contained in the class

    # self here is a way to refer to object of the same class and its attribute

    def __init__(self,name,balance,ac_num):

        self.Name=name
        self.AC_balance=balance
        self.AcNum = ac_num
```

now i just need to create one object for customers with these attributes and can seamlessly avoid mixing up objects for these attributes across customers

```
c1=customer('lalit',2000,'A3124')
```

```
c1.Name
```

```
'lalit'
```

```
c1.AC_balance
```

```
2000
```

```
c1.AcNum
```

```
'A3124'
```

I can also attach methods/function to this class which will be available to object of this class only

```
class customer():

    # the first function is __init__ , with double underscore
    # this is used to set attributes of object of the class customer when it gets created
    # we can also put data here which can be used by any object of the class customer
    # can also be used by other methods/functions contained in the class

    # self here is a way to refer to object of the same class and its attribute

    def __init__(self,name,balance,ac_num):

        self.Name=name
        self.AC_balance=balance
        self.AcNum = ac_num

    def withdraw(self,amount):

        self.AC_balance -= amount

    def get_account_number(self):

        print(self.AcNum)
```

```
c1=customer('lalit',2000,'A3124')
```

```
c1.AC_balance
```

```
2000
```

```
c1.withdraw(243)
```

```
c1.AC_balance
```

```
1757
```

```
c1.get_account_number()
```

```
A3124
```

As a final note to this discussion; If idea of classes seemed too daunting, you can safely skip this. You will never need to write to your own classes until you start to work with some bigger projects which implement their own algorithm or complex data processing routine.

In this module we are going to learn to handle datasets; creating datasets, reading from external files , modifying datasets. We will also see summarizing and visualizing in python with packages numpy, pandas and seaborn.

Chapter 2 : Data Handling with Python

In this module we'll discuss data handling with python. Discussion will be largely around two packages `numpy` and `pandas` . Numpy is the original package in python designed to work with multidimensional arrays which eventually enables us to work with data files. Pandas is a high level package written on top of numpy which makes the syntax much more easier so that we can focus on logic of data handling processes rather than getting bogged down with increasingly complex syntax of numpy. However numpy comes with lot of functions which we'll be using for data manipulation.

Since pandas is written on top of numpy, its good to know how numpy works in general to understand rationale behind lot of syntactical choices of pandas. Lets begin the discussion with numpy.

Numpy

Through numpy we will learn to create and handle arrays. Arrays set a background for handling columns in our datasets when we eventually move to pandas dataframes.
We will cover the following topics in Numpy:

- creating nd arrays
- subsetting with indices and conditions
- comparison with np and math functions [np.sqrt , log etc] and special numpy functions

For this course, we will consider only two dimensional arrays, though technically, we can create arrays with more than two dimensions.

We start with importing the package numpy giving it the alias np.

```
import numpy as np
```

We start with creating a 2 dimensional array and assign it to the variable 'b'. It is simply a list of lists.

```
b = np.array([[3,20,99],[-13,4.5,26],[0,-1,20],[5,78,-19]])
b
```

```
array([[ 3. , 20. , 99. ],
 [-13. ,  4.5,  26. ],
 [ 0. , -1. , 20. ],
 [ 5. , 78. , -19. ]])
```

We have passed 4 lists and each of the lists contains 3 elements. This makes 'b' a 2 dimensional array.

We can also determine the size of an array by using its shape attribute.

```
b.shape
```

```
(4, 3)
```

Each dimension in a numpy array is referred to by the argument 'axis'. 2 dimensional arrays have two axes, namely 0 and 1. Since there are two axes here, we will need to pass two indices when accessing the values in the numpy array. Numbering along both the axes starts with a 0.

```
b
```

```
array([[ 3. , 20. , 99. ],
   [-13. ,  4.5,  26. ],
   [ 0. , -1. , 20. ],
   [ 5. , 78. , -19. ]])
```

Assuming that we want to access the value -1 from the array 'b', we will need to access it with both its indices along the two axes.

```
b[2,1]
```

```
-1.0
```

Subset an array using Indexing and Slicing

In order to access -1, we will need to first pass index 2 which refers to the third list and then we will need to pass index 1 which refers to the second element in the third list. Axis 0 as refers to each list and axis 1 refers to elements present in each list. First index is the index of the list where the element is (i.e. 2) and the second index is its position within that list (i.e. 1).

Indexing and slicing here works just like it did in lists, only difference being that here we are considering 2 dimensions.

Now lets consider a few more examples

```
print(b)
b[:,1]
```

```
[[ 3.  20.  99.]
[-13.   4.5 26.]
[ 0.  -1.  20.]
[ 5.  78. -19.]]
```

```
array([20. ,  4.5, -1. , 78. ])
```

This statement gives us the second element from all the lists.

```
b[1,:]
```

```
array([-13. ,   4.5,  26. ])
```

The above statement gives us all the elements from the second list.

By default, we can access all the elements of a list by providing a single index as well. The above code can also be written as:

```
b[1]
```

```
array([-13. ,   4.5,  26. ])
```

We can access multiple elements of a 2 dimensional array by passing multiple indices as well.

```
print(b)
b[[0,1,1],[1,2,1]]
```

```
[[ 3.  20.  99.]
[-13.   4.5 26.]
[ 0.  -1.  20.]
[ 5.  78. -19.]]
```

```
array([20. , 26. ,  4.5])
```

Here, values are returned by pairing of indices i.e. (0,1), (1,2) and (1,1). We will get the first element when we run b[0,1] (i.e. the first list and second element within that list); the second element when we run b[1,2] (i.e. the second list and the third element within that list) and the third element when we run b[1,1] (i.e. the second list and the second element within that list). The three values returned in the array above can also be obtained by the three print statements written below:

```
print(b[0,1])
print(b[1,2])
print(b[1,1])
```

```
20.0
26.0
4.5
```

This way of accessing the index can be used for modification as well e.g. updating the values of those indices.

```
print(b)
b[[0,1,1],[1,2,1]]=[-10,-20,-30]
print(b)
```

```
[[ 3.  20.  99. ]
[-13.   4.5 26. ]
[ 0.  -1.  20. ]
[ 5.  78. -19. ]]
[[ 3. -10.  99.]
[-13. -30. -20.]
[ 0. -1. 20.]
[ 5. 78. -19.]]
```

Here you can see that for each of the indices accessed, we updated the corresponding values i.e. the values present for the indices (0,1), (1,2) and (1,1) were updated. In other words, 20.0, 26.0 and 4.5 were replaced with -10, -20 and -30 respectively.

Subset an array using conditions

```
b
```

```
array([[ 3., -10.,  99.],
 [-13., -30., -20.],
 [ 0.,  -1.,  20.],
 [ 5.,  78., -19.]])
```

```
b>0
```

```
array([[ True, False,  True],
 [False, False, False],
 [False, False,  True],
 [ True,  True, False]])
```

On applying a condition on the array 'b', we get an array with Boolean values; True where the condition was met, False otherwise. We can use these Boolean values, obtained through using conditions, for subsetting the array.

```
b[b>0]
```

```
array([ 3., 99., 20.,  5., 78.])
```

The above statement returns all the elements from the array 'b' which are positive.

Let's say we now want all the positive elements from the third list. Then we need to run the following code:

```
print(b)
b[2]>0
```

```
[[ 3. -10.  99.]
[-13. -30. -20.]
[ 0.  -1.  20.]
[ 5.  78. -19.]]
```

```
array([False, False,  True])
```

When we write b[2]>0, it returns a logical array, returning True wherever the list's value is positive and False otherwise.

Subsetting the list in the following way will, using the condition b[2]>0, will return the actual positive value.

```
b[2,b[2]>0]
```

```
array([20.])
```

Now, what if I want the values from all lists only for those indices where the values in the third list were either 0 or positive.

```
print(b)
print(b[2]>=0)
print(b[:,b[2]>=0])
```

```
[[ 3. -10.  99.]
[-13. -30. -20.]
[ 0.  -1.  20.]
[ 5.  78. -19.]]

[ True False  True]

[[ 3.  99.]
[-13. -20.]
[ 0.  20.]
[ 5. -19.]]
```

For the statement b[:,b[2]>=0], the ':' sign indicates that we are referring to all the lists and the condition 'b[2]>=0' would ensure that we will get the corresponding elements from all the lists which satisfy the condition that the third list is either 0 or positive. In other words, 'b[2]>=0' returns [True, False, True] which will enable us to get the first and the third values from all the lists.

Now lets consider the following scenario, where we want to apply the condition on the third element of each list and then apply the condition across all the elements of the lists:

```
b[:,2]>0
```

```
array([ True, False,  True, False])
```

Here, we are checking whether the third element in each list is positive or not. b[:,2]>0 returns a logical array. Note: it will have as many elements as the number of lists.

```
print(b)
print(b[:,2])
print(b[:,2]>0)
b[b[:,2]>0,:]
```

```
[[ 3. -10.  99.]
[-13. -30. -20.]
[ 0.  -1.  20.]
[ 5.  78. -19.]]

[ 99. -20.  20. -19.]
```

```
[ True False  True False]
```

```
array([[ 3., -10.,  99.],
       [ 0.,  -1.,  20.]])
```

Across the lists, it has extracted those values which correspond to the logical array. Using the statement print(b[:,2]>0), we see that only 99. and 20. are positive, i.e. the third element from each of the first and third lists are positive and hence True. On passing this condition to the array 'b', b[b[:,2]>0,:], we get all those lists wherever the condition evaluated to True i.e. the first and the third lists.

The idea of using numpy is that it allows us to apply functions on multiple values across a full dimension instead of single values. The math package on the other hand works on scalars of single values.

As an example, let's say we wanted to replace the entire 2nd list (index =1) with its absolute values.

We start with importing the math package.

```
import math as m
```

The function exp in the math package returns the exponential value of the number passed as argument.

```
x=-80
m.exp(x)
```

```
1.8048513878454153e-35
```

However, when we pass an array to this function instead of a single scalar value, we get an error.

```
b[1]
```

```
array([-13., -30., -20.])
```

```
b[1]=m.exp(b[1])
```

```
TypeError                                 Traceback (most recent call last)

 in
----> 1 b[1]=m.exp(b[1])
TypeError: only size-1 arrays can be converted to Python scalars
```

Basically, the math package converts its inputs to scalars, but since b[1] is an array of multiple elements, it gives an error.

We will need to use the corresponding numpy function to be able to return absolute values of arrays i.e. np.exp().

The following code will return the exponential values of the second list only.

```
print(b)
b[1]=np.exp(b[1])
print(b)
```

```
[[ 3. -10.  99.]
[-13. -30. -20.]
[ 0.  -1.  20.]
[ 5.  78. -19.]]
```

```
[[ 3.00000000e+00 -1.00000000e+01  9.90000000e+01]
 [ 2.26032941e-06  9.35762297e-14  2.06115362e-09]
 [ 0.00000000e+00 -1.00000000e+00  2.00000000e+01]
 [ 5.00000000e+00  7.80000000e+01 -1.90000000e+01]]
```

There are multiple such functions available in numpy. We can type 'np.' and press the 'tab' key to see the list of such functions.

All the functions present in the math package will be present in numpy package as well.

Reiterating the advantage of working with numpy instead of math package is that numpy enables us to work with complete arrays. We do not need to write a for loop to apply a function across the array.

axis argument

To understand the axis argument better, we will now explore the 'sum()' function which collapses the array.

```
np.sum(b)
```

```
175.00000226239064
```

Instead of summing the entire array 'b', we want to sum across the list i.e. axis = 0.

```
print(b)
np.sum(b,axis=0)
```

```
[[ 3.00000000e+00 -1.00000000e+01  9.90000000e+01]
 [ 2.26032941e-06  9.35762297e-14  2.06115362e-09]
 [ 0.00000000e+00 -1.00000000e+00  2.00000000e+01]
 [ 5.00000000e+00  7.80000000e+01 -1.90000000e+01]]
```

```
array([  8.00000226,  67.        , 100.        ])
```

If we want to sum all the elements of each list, then we will refer to axis = 1

```
np.sum(b,axis=1)
```

```
array([9.20000000e+01, 2.26239065e-06, 1.90000000e+01, 6.40000000e+01])
```

axis=0 here corresponds to elements across the lists , axis=1 corresponds to within the list elements.

Note: Pandas dataframes, which in a way are 2 dimensional numpy arrays, have each list in a numpy array correspond to a column in pandas dataframe. In a pandas dataframe, axis=0 would refer to rows and axis=1 would refer to columns.

Now we will go through some commonly used numpy functions. We will use the rarely used functions as and when we come across them.

The commonly used functions help in creating special kinds of numpy arrays.

arange()

```
np.arange(0,6)
```

```
array([0, 1, 2, 3, 4, 5])
```

The arange() function returns an array starting from 0 until (6-1) i.e. 5.

```
np.arange(2,8)
```

```
array([2, 3, 4, 5, 6, 7])
```

We can also control the starting and ending of an arange array. The above arange function starts from 2 and ends with (8-1) i.e. 7, incrementing by 1.

The arange function is used for creating a sequence of integers with different starting and ending points having an increment of 1.

linspace()

To create a more customized sequence we can use the linspace() function. The argument num gives the number of elements in sequence. The elements in the sequence will be equally spaced.

```
np.linspace(start=2,stop=10,num=15)
```

```
array([ 2.        ,  2.57142857,  3.14285714,  3.71428571,  4.28571429,
        4.85714286,  5.42857143,  6.        ,  6.57142857,  7.14285714,
        7.71428571,  8.28571429,  8.85714286,  9.42857143, 10.        ])
```

random.randint()

Given an array of numbers, we can randomly sample elements from that array using the randint() function from the random package.

```
np.random.randint(high=10,low=1,size=(2,3))
```

```
array([[4, 5, 8],
       [1, 3, 4]])
```

The above code creates a random array of size (2,3) i.e. two lists having three elements each. These random elements are chosen from numbers between 1 and 10.

random.random()

We can also create an array of random numbers using the random() function from the random package.

```
np.random.random(size=(3,4))
```

```
array([[0.83201385, 0.50934665, 0.04015334, 0.71826046],
       [0.25727761, 0.26890597, 0.71662683, 0.23890495],
       [0.17166143, 0.08512553, 0.43757489, 0.89178915]])
```

The above random() function creates an array of size (3,4) where the elements are real numbers between 0 to 1.

random.choice()

Consider an array ranging from 0 to 9:

```
x = np.arange(0, 10)
x
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
np.random.choice(x,6)
```

```
array([9, 0, 4, 3, 8, 5])
```

random.choice() functions helps us to select 6 random numbers from the input array x. Every time we run this code, the result will be different.

We can see that, at times, the function ends up picking up the same element twice. If we want to avoid that i.e. get each number only once or in other words, get the elements without replacement; then we need to set the argument 'replace' as False which is True by default.

```
np.random.choice(x,6,replace=False)
```

> array([4, 0, 7, 1, 8, 6])

Now when we run the above code again, we will get different values, but we will not see any number more than once.

We will also use the random.choice() function to simulate data often.

```
y = np.random.choice(['a','b'], 6)
print(y)
```

> ['b' 'a' 'b' 'b' 'a' 'b']

The code above picks 'a' or 'b' randomly 6 times.

```
y = np.random.choice(['a','b'], 1000)
```

The code above samples 'a' and 'b' a 1000 times. Now if we take unique values from this array, it will be 'a' and 'b', as shown by the code below. The return_counts argument gives the number of times the two elements are present in the array created.

```
np.unique(y, return_counts=True)
```

> (array(['a', 'b'], dtype='<U1'), array([509, 491], dtype=int64))

By default, both 'a' and 'b' get picked up with equal probability in the random sample. This does not mean that the individual values are not random; but the overall percentage of 'a' and 'b' remains almost the same.

However, if we want the two values according to a specific proportion, we can use the argument 'p' in the random.choice() function.

```
y=np.random.choice(['a','b'],1000,p=[0.8,0.2])
```

```
np.unique(y, return_counts=True)
```

> (array(['a', 'b'], dtype='<U1'), array([797, 203], dtype=int64))

Now we can observe that the value 'a' is present approximately 80% of the time and value 'b' appears around 20% of the time. the individual values are still random, though overall, 'a' will appear 80% of the time as specified and 'b' will appear 20% of the time. Since the underlying process is inherently random, these probabilities will be close to 80% and 20% respectively but may not be exactly the same. In fact, if we draw samples of smaller sizes, the difference could be quite wide as shown in the code below.

```
y=np.random.choice(['a','b'],10,p=[0.8,0.2])
np.unique(y, return_counts=True)
```

> (array(['a', 'b'], dtype='<U1'), array([9, 1], dtype=int64))

Here we sample only 10 values in the proportion 8:2. As you repeat this sampling process, at times the proportion may match, but many times the difference will be big. As we increase the size of the sample, the number of samples drawn for each element will be closer to the proportion specified.

sort()

To understand this function, lets create a single dimensional array:

```
x=np.random.randint(high=100,low=12,size=(15,))
print(x)
```

```
[89 60 31 82 97 92 14 96 75 12 36 37 25 27 62]
```

The array contains 15 random integers ranging from 12 to 100.

```
x.sort()
```

Sorting happens in place and it does not return anything.

```
print(x)
```

```
[12 14 25 27 31 36 37 60 62 75 82 89 92 96 97]
```

x.sort() sorts the entire array in an ascending order.

Now let us see how sorting works with 2 dimensional arrays.

```
x=np.random.randint(high=100,low=12,size=(4,6))
```

```
x
```

```
array([[61, 60, 78, 20, 56, 50],
       [27, 56, 88, 69, 40, 26],
       [35, 83, 40, 17, 74, 67],
       [33, 78, 25, 19, 53, 12]])
```

This array is 2 dimensional containing 4 lists and each list has 6 elements.

If we use the sort function directly, the correspondence between the elements is broken i.e. each individual list is sorted independent of the other lists. We may not want this. The first elements in each list belong together, so do the second and so on; but after sorting this correspondence is broken.

```
np.sort(x)
```

```
array([[20, 50, 56, 60, 61, 78],
       [26, 27, 40, 56, 69, 88],
       [17, 35, 40, 67, 74, 83],
       [12, 19, 25, 33, 53, 78]])
```

argsort()

In order to take care of this, we may use argsort() as follows:

For maintaining order along either of the axis, we can extract indices of the sorted values and reorder original array with these indices. Lets see the code below:

```
print(x)
x[:,2]
```

```
[[61 60 78 20 56 50]
 [27 56 88 69 40 26]
 [35 83 40 17 74 67]
 [33 78 25 19 53 12]]
```

```
array([78, 88, 40, 25])
```

These return the third element from each list of the 2 dimensional array x.

Lets say we want to sort the 2 dimensional array x by these values and the other values should move with them maintaining the correspondence. This is where we will use argsort() function.

```
print(x[:,2])
x[:,2].argsort()
```

```
[78 88 40 25]
```

```
array([3, 2, 0, 1], dtype=int64)
```

Instead of sorting the array, argsort() returns the indices of the elements after they are sorted.

The value with index 3 i.e. 25 should appear first, the value with index 2 i.e. 40 should appear next and so on.

We can now pass these indices to arrange all the lists according to them. We will observe that the correspondence does not break.

```
x[x[:,2].argsort(),:]
```

```
array([[33, 78, 25, 19, 53, 12],
       [35, 83, 40, 17, 74, 67],
       [61, 60, 78, 20, 56, 50],
       [27, 56, 88, 69, 40, 26]])
```

All the lists have been arranged according to order given by x[:,2].argsort() for the third element across the lists.

max() and argmax() - we can similarly use min() and argmin()

```
x=np.random.randint(high=100,low=12,size=(15,))
x
```

```
array([17, 77, 49, 36, 39, 27, 63, 99, 94, 22, 55, 66, 93, 32, 16])
```

```
x.max()
```

```
99
```

The max() function will simply return the maximum value from the array.

```
x.argmax()
```

```
7
```

The argmax() function on the other hand will simply give the index of the maximum value i.e. the maximum number 99 lies at the index 7.

Panda

In this section we will start with the python package named pandas which is primarily used for handling datasets in python.
We will cover the following topics:

1. Manually creating data frames
2. Reading data from external file
3. Peripheral summary of the data
4. Subsetting on the basis of row number and column number
5. Subsetting on the basis of conditions
6. Subsetting on the basis of column names

We start with importing the pandas package

```
import pandas as pd
import numpy as np
import random
```

1. Manually creating dataframes

There are two ways of creating dataframes:

1. From lists
2. From dictionary

1. Creating dataframe using lists:

We will start with creating some lists and then making a dataframe using these lists.

```
age=np.random.randint(low=16,high=80,size=[20,])
city=np.random.choice(['Mumbai','Delhi','Chennai','Kolkata'],20)
default=np.random.choice([0,1],20)
```

We can zip these lists to convert them to a single list tuples. Each tuple in the list will refer to a row in the dataframe.

```
mydata=list(zip(age,city,default))
mydata
```

```
[(33, 'Mumbai', 0),
(71, 'Kolkata', 1),
(28, 'Mumbai', 1),
(46, 'Mumbai', 1),
(22, 'Delhi', 1),
(23, 'Delhi', 0),
(58, 'Mumbai', 0),
(60, 'Kolkata', 1),
(69, 'Mumbai', 1),
(69, 'Kolkata', 1),
(34, 'Mumbai', 0),
(70, 'Delhi', 0),
(44, 'Chennai', 1),
(30, 'Delhi', 1),
(62, 'Delhi', 0),
(19, 'Mumbai', 1),
(68, 'Mumbai', 1),
(74, 'Chennai', 1),
(41, 'Mumbai', 1),
(24, 'Chennai', 1)]
```

Each of the tuples come from zipping the elements in each of the lists (age, city and default) that we created earlier.

Note: You may have different values when you run this code since we are randomly generating the lists using the random package.

We can then put this list of tuples in a dataframe simply by using the pd.DataFrame function.

```
df = pd.DataFrame(mydata, columns=["age", "city", "default"])
```

```
df.head() # we are using head() function which displays only the first 5 rows.
```

	age	city	default
0	33	Mumbai	0
1	71	Kolkata	1
2	28	Mumbai	1
3	46	Mumbai	1
4	22	Delhi	1

As you can observe, this is a simple dataframe with 3 columns and 20 rows, having the three lists: age, city and default as columns. The column names could have been different too, they do not have to necessarily match the list names.

2. Creating dataframe from a dictionary

Another way of creating dataframes is using a dictionary. Here we will need to provide the column names separately, which will be picked as the values of the key.

```
df = pd.DataFrame({"age":age, "city":city, "default":default})
```

Here the key values ("age", "city" and "default") will be taken as column names and the lists (age, city and default) will contain the values themselves.

```
df.head() # we are using head() function which displays only the first 5 rows.
```

	age	city	default
0	33	Mumbai	0
1	71	Kolkata	1
2	28	Mumbai	1
3	46	Mumbai	1
4	22	Delhi	1

In both the cases i.e. creating the dataframe using list or dictionary, the resultant dataframe is the same. The process of creating them is different but there is no difference in the resulting dataframes.

2. Reading data from external file

We can read data from external files using the pandas function read_csv().

Lets first create a string containing the path to the .csv file.

```
file=r'/Users/anjal/Dropbox/0.0 Data/loans data.csv'
```

Here, 'r' is added at the beginning of the path. This is to ensure that the file path is read as a raw string and any special character combinations are not interpreted as their special meaning by Python. e.g. \n means newline, which will be ignored from the file path when we add 'r' at the beginning of the string. otherwise it might lead to `Unicode` errors. Sometimes it will without putting `r` at the beginning of the path, but its a safer choice and make it a habit to always use this.

```
ld = pd.read_csv(file)
```

The pandas function read_csv() reads the file present in the path given by the agrument 'file'.

3. Peripheral summary of the data

```
ld.head()
# display of data in pdf will be truncated on the right hand side
```

	ID	Amount.Requested	Amount.Funded.By.Investors	Interest.Rate	Loan.Length	Loan.Purpo
0	81174.0	20000	20000	8.90%	36 months	debt_consoli
1	99592.0	19200	19200	12.12%	36 months	debt_consoli
2	80059.0	35000	35000	21.98%	60 months	debt_consoli
3	15825.0	10000	9975	9.99%	36 months	debt_consoli
4	33182.0	12000	12000	11.71%	36 months	credit_card

The head() function of the pandas dataframe created, by default, returns the top 5 rows of the dataframe. If we wish to see more or less rows, for instance 10 rows, then we will pass the number as an argument to the head() function.

```
ld.head(10)
# display of data in pdf will be truncated on the right hand side
```

	ID	Amount.Requested	Amount.Funded.By.Investors	Interest.Rate	Loan.Length	Loan.Purpo
0	81174.0	20000	20000	8.90%	36 months	debt_consoli
1	99592.0	19200	19200	12.12%	36 months	debt_consoli
2	80059.0	35000	35000	21.98%	60 months	debt_consoli
3	15825.0	10000	9975	9.99%	36 months	debt_consoli
4	33182.0	12000	12000	11.71%	36 months	credit_card
5	62403.0	6000	6000	15.31%	36 months	other
6	48808.0	10000	10000	7.90%	36 months	debt_consoli
7	22090.0	33500	33450	17.14%	60 months	credit_card
8	76404.0	14675	14675	14.33%	36 months	credit_card
9	15867.0	.	7000	6.91%	36 months	credit_card

Here, we get the top 10 rows of the dataframe.

We can get the column names by using the 'columns' attribute of the pandas dataframe.

```
ld.columns
```

```
Index(['ID', 'Amount.Requested', 'Amount.Funded.By.Investors', 'Interest.Rate',
  'Loan.Length', 'Loan.Purpose', 'Debt.To.Income.Ratio', 'State',
  'Home.Ownership', 'Monthly.Income', 'FICO.Range', 'Open.CREDIT.Lines',
  'Revolving.CREDIT.Balance', 'Inquiries.in.the.Last.6.Months',
  'Employment.Length'],
 dtype='object')
```

If we want to see the type of these columns, we can use the attribute 'dtypes' of the pandas dataframe.

```
ld.dtypes
```

```
ID                                float64
Amount.Requested                   object
Amount.Funded.By.Investors         object
Interest.Rate                      object
```

```
Loan.Length                         object
Loan.Purpose                        object
Debt.To.Income.Ratio                object
State                               object
Home.Ownership                      object
Monthly.Income                     float64
FICO.Range                          object
Open.CREDIT.Lines                   object
Revolving.CREDIT.Balance            object
Inquiries.in.the.Last.6.Months     float64
Employment.Length                   object
dtype: object
```

The float64 datatype refers to numeric columns and object datatype refers to categorical columns.

If we want a concise summary of the dataframe including information about null values, we use the info() function of the pandas dataframe.

```
ld.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2500 entries, 0 to 2499
Data columns (total 15 columns):
ID                                2499 non-null float64
Amount.Requested                  2499 non-null object
Amount.Funded.By.Investors        2499 non-null object
Interest.Rate                     2500 non-null object
Loan.Length                       2499 non-null object
Loan.Purpose                      2499 non-null object
Debt.To.Income.Ratio              2499 non-null object
State                             2499 non-null object
Home.Ownership                    2499 non-null object
Monthly.Income                    2497 non-null float64
FICO.Range                        2500 non-null object
Open.CREDIT.Lines                 2496 non-null object
Revolving.CREDIT.Balance          2497 non-null object
Inquiries.in.the.Last.6.Months    2497 non-null float64
Employment.Length                 2422 non-null object
dtypes: float64(3), object(12)
memory usage: 293.0+ KB
```

If we want to get the dimensions i.e. numbers of rows and columns of the data, we can use the attribute 'shape'.

```
ld.shape
```

> (2500, 15)

This dataframe has 2500 rows or observations and 15 columns.

4. Subsetting on the basis of row number and column number

We will now start with subsetting data on the basis of row and column numbers. The count starts with 0 for both rows and columns.

```
ld1=ld.iloc[3:7,1:5]
ld1
```

	Amount.Requested	Amount.Funded.By.Investors	Interest.Rate	Loan.Length
3	10000	9975	9.99%	36 months
4	12000	12000	11.71%	36 months
5	6000	6000	15.31%	36 months
6	10000	10000	7.90%	36 months

'iloc' refers to subsetting the dataframe by position. Here we have extracted the rows from 3rd to the 6th (7-1) position and columns from 1st to 4th (5-1) position.

To understand this further, we will further subset the 'ld1' dataframe. It currently has 4 rows and 4 columns. The indexes (3, 4, 5 and 6) come from the original dataframe. Lets subset the 'ld1' dataframe further.

```
ld1.iloc[2:4,1:3]
```

	Amount.Funded.By.Investors	Interest.Rate
5	6000	15.31%
6	10000	7.90%

You can see here that the positions are relative to the current dataframe 'ld1' and not the original dataframe 'ld'. Hence we end up with the 3rd and 4th rows along with 2nd and 3rd columns of the new dataframe 'ld1' and not the original dataframe 'ld'.

Generally, we do not subset dataframes by position. We normally subset the dataframes using conditions or column names.

5. & 6. Subsetting on the basis of conditions and columns

When we subset on the basis of conditions or column names, we can directly pass the conditions or column names in square brackets.

Lets say, we want to subset the dataframe and get only those rows for which the 'Home.Ownership' is of the type 'MORTGAGE' and the 'Monthly.Income' is above 5000.

Note: When we combine multiple conditions, we have to enclose them in parenthesis else your results will not be as expected.

```
ld[(ld['Home.Ownership']=='MORTGAGE') & (ld['Monthly.Income']>5000)].head()
# we are using head() function which displays only the first 5 rows.
# display of data in pdf will be truncated on the right hand side
```

	ID	Amount.Requested	Amount.Funded.By.Investors	Interest.Rate	Loan.Length	Loan.Purp
0	81174.0	20000	20000	8.90%	36 months	debt_consc
2	80059.0	35000	35000	21.98%	60 months	debt_consc
7	22090.0	33500	33450	17.14%	60 months	credit_card
12	41200.0	28000	27975	21.67%	60 months	debt_consc
20	86099.0	22000	21975	21.98%	36 months	debt_consc

On observing the results, you will notice that for each of the rows both the conditions will be satisfied i.e. 'Home.Ownership' will be 'MORTGAGE' and the 'Monthly.Income' will be greater than 5000.

In case we want to access a single columns data only, then we simply have to pass the column name in square brackets as follows:

```
ld['Home.Ownership'].head()
```

```
0    MORTGAGE
1    MORTGAGE
2    MORTGAGE
3    MORTGAGE
4        RENT
Name: Home.Ownership, dtype: object
```

However, if we want to access multiple columns, then the names need to be passed as a list. For instance, if we wanted to extract both 'Home.Ownership' and 'Monthly.Income', we would need to pass it as a list, as follows:

```
ld[['Home.Ownership','Monthly.Income']].head()
# note the double sqaure brackets used to subset the dataframe using multiple columns
```

	Home.Ownership	Monthly.Income
0	MORTGAGE	6541.67
1	MORTGAGE	4583.33
2	MORTGAGE	11500.00
3	MORTGAGE	3833.33
4	RENT	3195.00

If we intend to use both, condition as well as column names, we will need to use the .loc with the pandas dataframe name.

Observing the code below, we subset the dataframe using conditions and columns both. We are subsetting the rows, using the condition '(ld['Home.Ownership']=='MORTGAGE') & (ld['Monthly.Income']>5000)' and we extract the 'Home.Ownership' and 'Monthly.Income' columns.

Here, both the conditions should be met; to get that observation in the output. e.g. We can see in the first row of the dataframe that 'Home.Ownership' is 'MORTGAGE', the 'Monthly.Income' is more than 5000. If either of the condition is false, we will not see that observation in the resulting dataframe.

```
ld.loc[(ld['Home.Ownership']=='MORTGAGE') & (ld['Monthly.Income']>5000),['Home.Ownership','Monthly.Income']].head()
```

	Home.Ownership	Monthly.Income
0	MORTGAGE	6541.67
2	MORTGAGE	11500.00
7	MORTGAGE	13863.42
12	MORTGAGE	14166.67
20	MORTGAGE	6666.67

The resulting dataframe has only 2 columns and 686 rows. The rows correspond to the result obtained when the condition (ld['Home.Ownership']=='MORTGAGE') & (ld['Monthly.Income']>5000) is applied to the 'ld' dataframe.

Here, we have extracted rows on the basis of some conditions.

What if we wanted to subset only those rows which did not satisfy a condition i.e. we want to negate a condition. In order to do this, we can put a '~' sign before the condition.

In the following code, we will get all the observations that do not satisfy the condition `((ld['Home.Ownership']=='MORTGAGE') & (ld['Monthly.Income']>5000))`. Here, both the conditions may not be met; if either of them holds true, then we will get that observation in the output. e.g. Considering the first row, even though the 'Monthly.Income' is less than 5000 (i.e. one of the conditions is not met), since the other condition of 'Home.Ownership' being equal to 'MORTGAGE' holds true - the entire condition `((ld['Home.Ownership']=='MORTGAGE') & (ld['Monthly.Income']>5000))` is not negated and hence we see this observation in the output.

```
ld.loc[~((ld['Home.Ownership']=='MORTGAGE') & (ld['Monthly.Income']>5000)),
['Home.Ownership','Monthly.Income']].head()
```

	Home.Ownership	Monthly.Income
1	MORTGAGE	4583.33
3	MORTGAGE	3833.33
4	RENT	3195.00
5	OWN	4891.67
6	RENT	2916.67

In short, the '~' sign gives us rest of the observations by negating the condition.

Drop columns on the basis of names

To drop columns on the basis of column names, we can use the in-built drop() function.

In order to drop the columns, we pass the list of columns to be dropped along with specifying the axis argument as 1 (since we are dropping columns).

The following code will return all the columns except 'Home.Ownership' and 'Monthly.Income'.

```
ld.drop(['Home.Ownership','Monthly.Income'],axis=1).head()
# display of data in pdf will be truncated on the right hand side
```

	ID	Amount.Requested	Amount.Funded.By.Investors	Interest.Rate	Loan.Length	Loan.Purpo
0	81174.0	20000	20000	8.90%	36 months	debt_consoli
1	99592.0	19200	19200	12.12%	36 months	debt_consoli
2	80059.0	35000	35000	21.98%	60 months	debt_consoli
3	15825.0	10000	9975	9.99%	36 months	debt_consoli
4	33182.0	12000	12000	11.71%	36 months	credit_card

However, when we check the columns of the 'ld' dataframe now, the two columns which we presumably deleted, are still there.

```
ld.columns
```

```
Index(['ID', 'Amount.Requested', 'Amount.Funded.By.Investors', 'Interest.Rate',
  'Loan.Length', 'Loan.Purpose', 'Debt.To.Income.Ratio', 'State',
  'Home.Ownership', 'Monthly.Income', 'FICO.Range', 'Open.CREDIT.Lines',
  'Revolving.CREDIT.Balance', 'Inquiries.in.the.Last.6.Months',
  'Employment.Length'],
 dtype='object')
```

Basically, ld.drop(['Home.Ownership','Monthly.Income'],axis=1) did not change the original dataframe and the deleted columns are still present in 'ld' dataframe.

What happens is that the ld.drop() function gives us an output; it does not make any inplace changes.

So, in case we wish to delete the columns from the original dataframe, we can do two things:

- Equate the output to the original dataframe
- Set the 'inplace' argument of the drop() function to True

We can update the original dataframe by equating the output to the original dataframe as follows:

```
ld=ld.drop(['Home.Ownership','Monthly.Income'],axis=1)
ld.columns
```

```
Index(['ID', 'Amount.Requested', 'Amount.Funded.By.Investors', 'Interest.Rate',
 'Loan.Length', 'Loan.Purpose', 'Debt.To.Income.Ratio', 'State',
 'FICO.Range', 'Open.CREDIT.Lines', 'Revolving.CREDIT.Balance',
 'Inquiries.in.the.Last.6.Months', 'Employment.Length'],
dtype='object')
```

The second way to update the original dataframe is to set the 'inplace' argument of the drop() function to True

```
ld.drop(['State'],axis=1,inplace=True)
ld.columns
```

```
Index(['ID', 'Amount.Requested', 'Amount.Funded.By.Investors', 'Interest.Rate',
 'Loan.Length', 'Loan.Purpose', 'Debt.To.Income.Ratio', 'FICO.Range',
 'Open.CREDIT.Lines', 'Revolving.CREDIT.Balance',
 'Inquiries.in.the.Last.6.Months', 'Employment.Length'],
dtype='object')
```

Now you will notice that the deleted columns are not present in the original dataframe 'ld' anymore.

We need to be careful when using the inplace=True option; the function drop() doesn't output anything. So we should not equate ld.drop(['State'],axis=1,inplace=True) to the original dataframe. If we equate it to the original dataframe 'ld', then 'ld' will end up as a None type object.

Drop columns using del keyword

We can also delete a column using the 'del' keyword. The following code will remove the column 'Employment.Length' from the original dataframe 'ld'.

```
del ld['Employment.Length']
ld.columns
```

```
Index(['ID', 'Amount.Requested', 'Amount.Funded.By.Investors', 'Interest.Rate',
 'Loan.Length', 'Loan.Purpose', 'Debt.To.Income.Ratio', 'FICO.Range',
 'Open.CREDIT.Lines', 'Revolving.CREDIT.Balance',
 'Inquiries.in.the.Last.6.Months'],
dtype='object')
```

On checking the columns of the 'ld' dataframe, we can observe that 'Employment.Length' column is not present.

Modifying data with Pandas

Now we will be covering how to modify dataframes using pandas. Note: We will be using a simulated dataset to understand this, as all data pre processing opportunities might not be present in a single real world dataset. However, the various techniques that we'll learn here will be useful in different datasets that we come across. Some of the techniques that we cover are as follows:

1. Changing variable types

2. Adding/modifying variables with algebraic operations
3. Adding/modifying variables based on conditions
4. Handling missing values
5. Creating flag variables
6. Creating multiple columns from a variable separated by a delimiter

Lets start with importing the packages we need:

```
import numpy as np
import pandas as pd
```

We will start with creating a custom dataframe having 7 columns and 50 rows as follows:

```
age=np.random.choice([15,20,30,45,12,'10',15,'34',7,'missing'],50)
fico=np.random.choice(['100-150','150-200','200-250','250-300'],50)
city=np.random.choice(['Mumbai','Delhi','Chennai','Kolkata'],50)
ID=np.arange(50)
rating=np.random.choice(['Excellent','Good','Bad','Pathetic'],50)
balance=np.random.choice([10000,20000,30000,40000,np.nan,50000,60000],50)
children=np.random.randint(high=5,low=0,size=(50,))
```

```
mydata=pd.DataFrame({'ID':ID,'age':age,'fico':fico,'city':city,'rating':rating,'balance':balance,'children':children})
```

```
mydata.head()
# data displace in pdf will be truncated on right hand side
```

	ID	age	fico	city	rating	balance	children
0	0	12	250-300	Chennai	Excellent	10000.0	3
1	1	15	150-200	Chennai	Bad	20000.0	3
2	2	missing	250-300	Chennai	Pathetic	20000.0	2
3	3	45	250-300	Delhi	Bad	NaN	3
4	4	missing	250-300	Delhi	Pathetic	50000.0	2

1. Changing variable types

Lets check the datatypes of the columns in the mydata dataframe.

```
mydata.dtypes
```

```
ID            int32
age          object
fico         object
city         object
rating       object
balance     float64
children      int32
dtype: object
```

We can see that 'age' column is of the object datatype, though it should have been numeric, maybe due to some character values in the column. We can change the datatype to 'numeric'; the character values which cannot be changed to numeric will be assigned missing values i.e. NaN's automatically.

There are multiple numeric formats in Python e.g. integer, float, unsigned integer etc. The to_numeric() functions chooses the best one for the column under consideration.

```
mydata['age']=pd.to_numeric(mydata['age'])
```

> ValueError Traceback (most recent call last)
>
> pandas/*libs/src\inference.pyx in pandas*.libs.lib.maybe_convert_numeric()
>
> ValueError: Unable to parse string "missing"

When we run the code above, we get an error i.e. "Unable to parse string "missing" at position 2". This error means that there are a few values in the column that cannot be converted to numbers; in our case its the value 'missing' which cannot be converted to a number. In order to handle this, we need to set the errors argument of the to_numeric() function to 'coerce' i.e. errors='coerce'. When we use this argument, wherever it was not possible to convert the values to numeric, it converted them to missing values i.e. NaN's.

```
mydata['age']=pd.to_numeric(mydata['age'], errors='coerce')
mydata['age'].head()
```

> 0 12.0
> 1 15.0
> 2 NaN
> 3 45.0
> 4 NaN
> Name: age, dtype: float64

As we can observe in the rows 2,4,etc, wherever there was the 'missing' string present, which could not be converted to numbers are now converted to NaN's or missing values.

2. Adding/modifying variables with algebraic operations

Now lets look at some additions and modifications of columns using algebraic operations.

Adding a column with constant value

```
mydata['const_var']=100
```

The above code adds a new column 'const_var' to the mydata dataframe and each element in that column is 100.

```
mydata.head()
```

	ID	age	fico	city	rating	balance	children	const_var
0	0	12.0	250-300	Chennai	Excellent	10000.0	3	100
1	1	15.0	150-200	Chennai	Bad	20000.0	3	100
2	2	NaN	250-300	Chennai	Pathetic	20000.0	2	100
3	3	45.0	250-300	Delhi	Bad	NaN	3	100
4	4	NaN	250-300	Delhi	Pathetic	50000.0	2	100

Apply function on entire columns

If we want to apply a function on an entire column of a dataframe, we use a numpy function; e.g log as shown below:

```
mydata['balance_log']=np.log(mydata['balance'])
```

The code above creates a new column 'balance_log' which has the logarithmic value of each element present in the 'balance' column. A numpy function np.log() is used to do this.

```
mydata.head()
```

	ID	age	fico	city	rating	balance	children	const_var	balance_log
0	0	12.0	250-300	Chennai	Excellent	10000.0	3	100	9.210340
1	1	15.0	150-200	Chennai	Bad	20000.0	3	100	9.903488
2	2	NaN	250-300	Chennai	Pathetic	20000.0	2	100	9.903488
3	3	45.0	250-300	Delhi	Bad	NaN	3	100	NaN
4	4	NaN	250-300	Delhi	Pathetic	50000.0	2	100	10.819778

Add columns using complex algebraic calculations

We can do many complex algebraic calculations as well to create/add new columns to the data.

```
mydata['age_children_ratio']=mydata['age']/mydata['children']
```

The code above creates a new column 'age_children_ratio'; each element of which will be the result of the division of the corresponding elements present in the 'age' and 'children' columns.

```
mydata.head(10)
# display in pdf will be truncated on right hand side
```

	ID	age	fico	city	rating	balance	children	const_var	balance_log	age_children_rat
0	0	12.0	250-300	Chennai	Excellent	10000.0	3	100	9.210340	4.000000
1	1	15.0	150-200	Chennai	Bad	20000.0	3	100	9.903488	5.000000
2	2	NaN	250-300	Chennai	Pathetic	20000.0	2	100	9.903488	NaN
3	3	45.0	250-300	Delhi	Bad	NaN	3	100	NaN	15.000000
4	4	NaN	250-300	Delhi	Pathetic	50000.0	2	100	10.819778	NaN
5	5	20.0	100-150	Delhi	Pathetic	60000.0	3	100	11.002100	6.666667
6	6	34.0	100-150	Mumbai	Good	40000.0	0	100	10.596635	inf
7	7	20.0	200-250	Kolkata	Pathetic	30000.0	4	100	10.308953	5.000000
8	8	20.0	150-200	Mumbai	Good	20000.0	3	100	9.903488	6.666667
9	9	20.0	150-200	Chennai	Bad	50000.0	4	100	10.819778	5.000000

Notice that when a missing value is involved in any calculation, the result is also a missing value. We observe that in the 'age_children_ratio' column we have both NaN's (missing values) as well as inf (infinity). We get missing values in the 'age_children_ratio' column wherever 'age' has missing values and we get 'inf' wherever the number of children is 0 and we end up dividing by 0.

Handling missing values

Lets say we did not want missing values involved in the calculation i.e. we want to impute the missing values before computing the 'age_children_ratio' column. For this we would first need to identify the missing values. The isnull() function will give us a logical array which can be used to isolate missing values and update these with whatever value we want to impute with.

```
mydata['age'].isnull().head()
```

```
0   False
1   False
2    True
3   False
4    True
Name: age, dtype: bool
```

In the outcome of the code above, we observe that wherever there is a missing value the corresponding logical value is True.

If we want to know the number of missing values, we can sum the logical array as follows:

```
mydata['age'].isnull().sum()
```

```
5
```

We have 5 missing values in the 'age' column.

The following code returns only those elements where the 'age' column has missing values.

```
mydata.loc[mydata['age'].isnull(),'age']
```

```
2   NaN
4   NaN
17  NaN
36  NaN
38  NaN
Name: age, dtype: float64
```

One of the ways of imputing the missing values is with mean. Once these values are imputed, we then carry out the calculation done above.

```
mydata.loc[mydata['age'].isnull(),'age'] = np.mean(mydata['age'])
```

In the code above, using the loc() function of the dataframe, on the row side we first access those rows where the 'age' column is null and on the column side we access only the 'age' column. In other words, all the missing values from the age column will be replaced with the mean computed using the 'age' column.

```
mydata['age'].head()
```

```
0   12.000000
1   15.000000
2   19.533333
3   45.000000
4   19.533333
Name: age, dtype: float64
```

The missing values from the 'age' column has been replaced by the mean of the column i.e. 19.533333.

Now, we can compute the 'age_children_ratio' again; this time without missing values. We will observe that there are no missing values in the newly created column now. We however, observe inf's i.e. infinity which occurs wherever we divide by 0.

```
mydata['age_children_ratio']=mydata['age']/mydata['children']
```

```
mydata.head()
#display in pdf will be truncated on the right hand side
```

	ID	age	fico	city	rating	balance	children	const_var	balance_log	age_childre
0	0	12.000000	250-300	Chennai	Excellent	10000.0	3	100	9.210340	4.000000
1	1	15.000000	150-200	Chennai	Bad	20000.0	3	100	9.903488	5.000000
2	2	19.533333	250-300	Chennai	Pathetic	20000.0	2	100	9.903488	9.766667
3	3	45.000000	250-300	Delhi	Bad	NaN	3	100	NaN	15.000000
4	4	19.533333	250-300	Delhi	Pathetic	50000.0	2	100	10.819778	9.766667

3. Adding/modifying variables based on conditions

Now we will see how do we add or modify variables based on conditions.

Lets say we want to replace the 'rating' column values with some numeric score - {'pathetic' : -1 , 'bad' : 0 , 'good or excellent': 1}. We can do it using the np.replace() function as follows:

```
mydata['rating_score']=np.where(mydata['rating'].isin(['Good','Excellent']),1,0)
```

Using the above code, we create a new column 'rating_score' and wherever either 'Good' or 'Excellent' is present, we replace it with a 1 else with a 0 as we can see below. The function isin() is used when we need to consider multiple values; in our case 'Good' and 'Excellent'.

```
mydata.head()
# display in pdf will be truncated on the right hand side
```

	ID	age	fico	city	rating	balance	children	const_var	balance_log	age_childre
0	0	12.000000	250-300	Chennai	Excellent	10000.0	3	100	9.210340	4.000000
1	1	15.000000	150-200	Chennai	Bad	20000.0	3	100	9.903488	5.000000
2	2	19.533333	250-300	Chennai	Pathetic	20000.0	2	100	9.903488	9.766667
3	3	45.000000	250-300	Delhi	Bad	NaN	3	100	NaN	15.000000
4	4	19.533333	250-300	Delhi	Pathetic	50000.0	2	100	10.819778	9.766667

An alternative way of updating the 'rating' column will be as follows:

```
mydata.loc[mydata['rating']=='Pathetic','rating_score']=-1
```

In the code above, wherever the value in 'rating' column is 'Pathetic', we update the 'rating_score' column to -1 and leave the rest as is. The above code could be been written using np.where() function as well. The np.where() function is similar to the ifelse statement we may have seen in other languages.

```
mydata.head()
#display in the pdf will be truncated on the right hand side
```

	ID	age	fico	city	rating	balance	children	const_var	balance_log	age_childre
0	0	12.000000	250-300	Chennai	Excellent	10000.0	3	100	9.210340	4.000000
1	1	15.000000	150-200	Chennai	Bad	20000.0	3	100	9.903488	5.000000
2	2	19.533333	250-300	Chennai	Pathetic	20000.0	2	100	9.903488	9.766667
3	3	45.000000	250-300	Delhi	Bad	NaN	3	100	NaN	15.000000
4	4	19.533333	250-300	Delhi	Pathetic	50000.0	2	100	10.819778	9.766667

Now we can see that the 'rating_score' column takes the values 0, 1 and -1 depending on the corresponding values from the 'rating' column.

At times, we may have columns which we may want to split into multiple columns; column 'fico' in our case. One sample value is '100-150'. The datatype for 'fico' is considered as object. Its a difficult problem to solve if we use a for loop for processing each value. However, we will discuss an easier approach which will be very useful to know when pre-processing your data.

Coming back to the 'fico' column, one of the first thing that comes to mind when we want to separate the values from 'fico' column into multiple columns is the split() function. The split function works on strings, but the current datatype of 'fico' column is object; object type does not understand string functions. If we apply the split() function directly on 'fico', we will get the following error.

```
mydata['fico'].split()
```

```
AttributeError                Traceback (most recent call last)
⋮
AttributeError: 'Series' object has no attribute 'split'
```

In order to handle this, we will first need to extract the 'str' attribute of the 'fico' column and then apply the split() function. This will be the case for all string functions and not just split().

```
mydata['fico'].str.split("-").head()
```

```
0    [250, 300]
1    [150, 200]
2    [250, 300]
3    [250, 300]
4    [250, 300]
Name: fico, dtype: object
```

We can see that each of the elements have been split on the basis of '-'. However, it is still present in a single column. We need the values in separate columns. We will set the 'expand' argument of the split() function to True in order to handle this.

```
mydata['fico'].str.split("-",expand=True).head()
```

	0	1
0	250	300
1	150	200
2	250	300
3	250	300
4	250	300

This code by default creates a dataframe.

```
k=mydata['fico'].str.split("-",expand=True).astype(float)
```

Notice that we have converted the columns to float using the astype(float) function; since after splitting, by default, the datatype of each column created would be object. But we want to consider each column as numeric datatype, hence the columns are converted to float. Converting to float is not a required step when splitting columns. We do it only because these values are supposed to be considered numeric in the current context.

We can now access the individual columns using k[0] or k[1].

```
k[0].head()
```

```
0    250.0
1    150.0
2    250.0
3    250.0
4    250.0
Name: 0, dtype: float64
```

We can either concatenate this dataframe to the 'mydata' dataframe after giving proper header to both the columns or we can directly assign two new columns in the 'mydata' dataframe as follows:

```
mydata['f1'],mydata['f2']=k[0],k[1]
```

```
mydata.head()
# display in pdf will be truncated on the right hand side
```

	ID	age	fico	city	rating	balance	children	const_var	balance_log	age_childre
0	0	12.000000	250-300	Chennai	Excellent	10000.0	3	100	9.210340	4.000000
1	1	15.000000	150-200	Chennai	Bad	20000.0	3	100	9.903488	5.000000
2	2	19.533333	250-300	Chennai	Pathetic	20000.0	2	100	9.903488	9.766667
3	3	45.000000	250-300	Delhi	Bad	NaN	3	100	NaN	15.000000
4	4	19.533333	250-300	Delhi	Pathetic	50000.0	2	100	10.819778	9.766667

We do not need the 'fico' column anymore as we have its values in tow separate columns; hence we will delete it.

```
del mydata['fico']
```

```
mydata.head()
# display in pdf will be truncated on the right hand side
```

	ID	age	city	rating	balance	children	const_var	balance_log	age_children_ratio
0	0	12.000000	Chennai	Excellent	10000.0	3	100	9.210340	4.000000
1	1	15.000000	Chennai	Bad	20000.0	3	100	9.903488	5.000000
2	2	19.533333	Chennai	Pathetic	20000.0	2	100	9.903488	9.766667
3	3	45.000000	Delhi	Bad	NaN	3	100	NaN	15.000000
4	4	19.533333	Delhi	Pathetic	50000.0	2	100	10.819778	9.766667

Converting categorical variables to flag variables or dummies

For all the machine algorithm that we will encounter, we will need to convert categorical variables to flag variables. If a variable has n distinct categories, we will need to create (n-1) flag variables. We cannot use categorical variables - character type variables - object type variables as is in the data that we pass to the machine learning algorithms.

Manually, dummies can be created as follows:

Lets consider the city variable.

```
print(mydata['city'].unique())
print(mydata['city'].nunique())
```

```
['Chennai' 'Delhi' 'Mumbai' 'Kolkata']
4
```

It consists of 4 unique elements - 'Kolkata', 'Mumbai', 'Chennai', 'Delhi'. For this variable, we will need to create three dummies.

The code below creates a flag variable when the 'city' column has the value 'Mumbai'.

```
mydata['city_mumbai']=np.where(mydata['city']=='Mumbai',1,0)
```

Wherever the variable 'city' takes the value 'Mumbai', the flag variable 'city_mumbai' will be 1 otherwise 0.

There is another way to do this, where we write the condition and convert the logical value to integer.

```
mydata['city_chennai']=(mydata['city']=='Chennai').astype(int)
```

Code "mydata['city']=='Chennai'" gives a logical array; wherever the city is 'Chennai', the value on 'city_chennai' flag variable is True, else False.

```
(mydata['city']=='Chennai').head()
```

```
0    True
1    True
2    True
3   False
```

```
4    False
Name: city, dtype: bool
```

When we convert it to an integer, wherever there was True, we get a 1 and wherever there was False, we get a 0.

```
((mydata['city']=='Chennai').astype(int)).head()
```

```
0    1
1    1
2    1
3    0
4    0
Name: city, dtype: int32
```

We can use either of the methods for creating flag variables, the end result is same.

We need to create one more flag variable.

```
mydata['city_kolkata']=np.where(mydata['city']=='Kolkata',1,0)
```

Once the flag variables have been created, we do not need the original variable i.e. we do not need the 'city' variable anymore.

```
del mydata['city']
```

This way of creating dummies requires a lot of coding, even if we somehow use a for loop. As an alternative, we can use get_dummies() function from pandas directly to do this.

We will create dummies for the variable 'rating' using this method.

```
print(mydata['rating'].unique())
print(mydata['rating'].nunique())
```

```
['Excellent' 'Bad' 'Pathetic' 'Good']
4
```

This too has 4 unique values.

The pandas function which creates dummies is get_dummies() in which we pass the column for which dummies need to be created. By default, the get_dummies function creates n dummies if n unique values are present in the column i.e. for 'rating' column, by default, get_dummies function, creates 4 dummy variables. We do not want that. Hence, we pass the argument 'drop_first=True' which removes one of the dummies and creates (n-1) dummies only. Setting the 'prefix' argument helps to identify which columns are the dummy variables created for. It does this by adding whatever string you give the 'prefix' argument as a prefix to each of the dummy variables created. In our case 'rating_' will be appended to each dummy variable as a prefix.

```
dummy=pd.get_dummies(mydata['rating'],drop_first=True,prefix='rating')
```

The get_dummies() function has created a column for 'Excellent', 'Good' and 'Pathetic' but has dropped the column for 'Bad'.

```
dummy.head()
```

	rating_Excellent	rating_Good	rating_Pathetic
0	1	0	0
1	0	0	0
2	0	0	1
3	0	0	0
4	0	0	1

We can now simply attach these columns to the data using pandas concat() function.

```
mydata=pd.concat([mydata,dummy],axis=1)
```

The concat() function will take the axis argument as 1 since we are attaching the columns.

After creating dummies for 'rating', we will now drop the original column.

```
del mydata['rating']
```

```
mydata.columns
```

```
Index(['ID', 'age', 'balance', 'children', 'const_var', 'balance_log',
  'age_children_ratio', 'rating_score', 'f1', 'f2', 'city_mumbai',
  'city_chennai', 'city_kolkata', 'rating_Excellent', 'rating_Good',
  'rating_Pathetic'],
 dtype='object')
```

We need to keep in mind that we will not be doing all of what we learned here at once for any one dataset. Some of these techniques will be useful at a time while preparing data for machine learning algorithms.

Sorting and Merging Dataframes

Here will will cover the following:

1. Sorting data by one or more column(s) in ascending or descending manner
2. Combining dataframes vertically or horizontally
3. Merging dataframes using keys as well as left, right, inner and outer joins

Lets start with importing pandas and numpy.

```
import numpy as np
import pandas as pd
```

We will create a dataframe with random values.

```
df = pd.DataFrame(np.random.randint(2, 8,(20,4)), columns=['A','B','C','D'])
```

The random.randint() function creates 4 columns having 20 rows with values ranging between 2 and 8.

```
df.head()
```

	A	B	C	D
0	4	7	4	6
1	2	7	5	6
2	2	3	5	7
3	4	6	2	4
4	6	5	4	4

1. Sorting data by one or more column(s) in ascending or descending manner

If we wish to sort the dataframe by column A, we can do that using the sort_values() function on the dataframe.

```
df.sort_values("A").head()
```

	A	B	C	D
9	2	5	6	7
14	2	6	4	6
18	2	7	7	3
6	2	4	2	5
19	2	6	4	6

The output that we get is sorted by the column 'A'. But when we type 'df' again to view the dataframe, we see that there are not changes; df is the same as it was before sorting.

```
df.head()
```

	A	B	C	D
0	4	7	4	6
1	2	7	5	6
2	2	3	5	7
3	4	6	2	4
4	6	5	4	4

To handle this, there are two things that we can do:

- assign the sorted dataframe to the original dataframe
- use 'inplace=True' argument when sorting using sort_values() function

```
df = df.sort_values("A") # assign the sorted dataframe to the original dataframe
```

```
df.sort_values("A", inplace=True) # use 'inplace=True' argument when sorting using sort_values() function
```

Now when we observe the dataframe 'df', it will be sorted by 'A' in an ascending manner.

```
df.head()
```

	A	B	C	D
9	2	5	6	7
14	2	6	4	6
18	2	7	7	3
6	2	4	2	5
19	2	6	4	6

In case we wish to sort the dataframe in a descending manner, we can set the argument ascending=False in the sort_values() function.

```
df.sort_values("A", inplace=True, ascending=False)
```

Now the dataset will be sorted in the reverse order of the values of 'A'.

```
df.head()
```

	A	B	C	D
17	7	7	5	5
10	7	7	4	4
15	7	4	6	7
4	6	5	4	4
11	6	3	7	2

We can sort the dataframes by multiple columns also.

Sorting by next column in the sequence happens within the groups formed after sorting of the previous columns.

In the code below, we can see that the 'ascending' argument takes values [True, False]. It is passed in the same order as the columns ['B','C']. This means that the column 'B' will be sorted in an ascending order and within the groups created by column 'B', column 'C' will be sorted in a descending order.

```
df.sort_values(['B','C'],ascending=[True,False]).head(10)
```

	A	B	C	D
12	5	2	3	5
11	6	3	7	2
13	3	3	7	6
2	2	3	5	7
7	5	3	3	4
15	7	4	6	7
16	4	4	4	2
6	2	4	2	5
9	2	5	6	7
4	6	5	4	4

We can observe that the column 'B' is sorted in an ascending order. Within the groups formed by column 'B', column 'C' sorts its values in descending order.

Although we have not taken an explicit example for character data, in case of character data, sorting happens in the order in which it is present in the dictionary.

3. Combining dataframes vertically or horizontally

Now we will see how to combine dataframes horizontally or vertically by stacking them.

Lets start by creating two dataframes.

```
df1 = pd.DataFrame([['a', 1], ['b', 2]],columns=['letter', 'number'])
df2 = pd.DataFrame([['c', 3, 'cat'], ['d', 4, 'dog']],columns=['letter', 'number',
'animal'])
```

```
df1
```

	letter	number
0	a	1
1	b	2

```
df2
```

	letter	number	animal
0	c	3	cat
1	d	4	dog

In order to combine these dataframes, we will use the concat() function of the pandas library.

The argument 'axis=0' combines the dataframes row-wise i.e. stacks the dataframes vertically.

```
pd.concat([df1,df2], axis=0)
```

Notice that the index of the two dataframes is not generated afresh in the concatenated dataframe. The original indices are stacked, so we end up with duplicate index names. More often than not, we would not want the indices to be stacked. We can avoid doing this by setting the 'ignore_index' argument to True in the concat() function.

```
pd.concat([df1,df2], axis=0, ignore_index=True)
```

	animal	letter	number
0	NaN	a	1
1	NaN	b	2
2	cat	c	3
3	dog	d	4

We can see that the index is generated afresh.

We discussed how the dataframes can be stacked vertically. Now lets see how they can be stacked horizontally.

Lets create one more dataframe.

```
df3 = pd.DataFrame([['bird', 'polly'], ['monkey', 'george'],
['tiger','john']],columns=['animal', 'name'])
```

In order to stack the dataframes column-wise i.e. horizontally, we will need to set the 'axis' argument to 1.

The datasets which we will stack horizontally are 'df1' and 'df3'.

```
df1
```

	letter	number
0	a	1
1	b	2

```
df3
```

	animal	name
0	bird	polly
1	monkey	george
2	tiger	john

```
pd.concat([df1,df3],axis=1)
```

	letter	number	animal	name
0	a	1.0	bird	polly
1	b	2.0	monkey	george
2	NaN	NaN	tiger	john

We see that when we use the concat() function with 'axis=1' argument, we combine the dataframes column-wise.

Since 'df3' dataframe has three rows, whereas 'df1' dataframe has only two rows, the remaining values are set to missing as can be observed in the dataframe above.

Many times our datasets need to be combined by keys instead of simply stacking them vertically or horizontally. As an example, lets consider the following dataframes:

```
df1=pd.DataFrame({"custid":[1,2,3,4,5],
                  "product":["Radio","Radio","Fridge","Fridge","Phone"]})
df2=pd.DataFrame({"custid":[3,4,5,6,7],
                  "state":["UP","UP","UP","MH","MH"]})
```

```
df1
```

	custid	product
0	1	Radio
1	2	Radio
2	3	Fridge
3	4	Fridge
4	5	Phone

```
df2
```

	custid	state
0	3	UP
1	4	UP
2	5	UP
3	6	MH
4	7	MH

Dataframe 'df1' contains information about the customer id and the product they purchased and dataframe 'df2' also contains the customer id along with which state they come from.

Notice that the first row of the two dataframes have different customer ids i.e. the first row contains information about different customers, hence it won't make sense to combine the two dataframes together horizontally.

In order to combine data from the two dataframes, we will first need to set a correspondence using customer id i.e. combine only those rows having a matching customer id and ignore the rest. In some situations, if there is data in one dataframe which is not present in the other dataframe, missing data will be filled in.

There are 4 ways in which the dataframes can be merged - inner join, outer join, left join and right join:

In the following code, we are joining the two dataframes 'df1' and 'df2' and the key or correspondence between the two dataframes is determined by 'custid' i.e. customer id. We use the inner join here (how='inner'), which retains only those rows which are present in both the dataframes. Since customer id's 3, 4 and 5 are common in both the dataframes, these three rows are returned as a result of the inner join along with corresponding information 'product' and 'state' from both the dataframes.

```
pd.merge(df1,df2,on=['custid'],how='inner')
```

	custid	product	state
0	3	Fridge	UP
1	4	Fridge	UP
2	5	Phone	UP

Now lets consider outer join. In outer join, we keep all the ids, starting at 1 and going up till 7. This leads to having missing values in some columns e.g. customer ids 6 and 7 were not present in the dataframe 'df1' containing product information. Naturally the product information for those customer ids will be absent.

Similarly, customer ids 1 and 2 were not present in the dataframe 'df2' containing state information. Hence, state information was missing for those customer ids.

Merging cannot fill in the data on its own if that information is not present in the original dataframes. We will explicitly see a lot of missing values in outer join.

```
pd.merge(df1,df2,on=['custid'],how='outer')
```

	custid	product	state
0	1	Radio	NaN
1	2	Radio	NaN
2	3	Fridge	UP
3	4	Fridge	UP
4	5	Phone	UP
5	6	NaN	MH
6	7	NaN	MH

Using the left join, we will see all customer ids present in the left dataframe 'df1' and only the corresponding product and state information from the two dataframes. The information present only in the right dataframe 'df2' is ignored i.e. customer ids 6 and 7 are ignored.

```
pd.merge(df1,df2,on=['custid'],how='left')
```

	custid	product	state
0	1	Radio	NaN
1	2	Radio	NaN
2	3	Fridge	UP
3	4	Fridge	UP
4	5	Phone	UP

Similarly, right join will contain all customer ids present in the right dataframe 'df2' irrespective of whether they are there in the left dataframe 'df1' or not.

```
pd.merge(df1,df2,on=['custid'],how='right')
```

	custid	product	state
0	3	Fridge	UP
1	4	Fridge	UP
2	5	Phone	UP
3	6	NaN	MH
4	7	NaN	MH

Numeric and Visual Summary of data

Here we will focus on how to look at summary of our data. We will observe how each variable in the data behaves individually and with other variables. We will understand how to get these summary statistics in Python.

We start with importing the numpy and pandas libraries.

```
import pandas as pd
import numpy as np
```

We will make use of bank-full.csv file available on LMS.

```
file=r'/Users/anjal/Dropbox/PDS V3/Data/bank-full.csv'
bd=pd.read_csv(file,delimiter=';')
```

Numeric summary of data

The first function we learn about is called describe(). It gives us a numerical summary of numeric variables. It returns 8 summary statistics i.e. count of non-missing values, mean, standard deviation, minimum, 25 percentile value, 50 percentile value, 75 percentile value and the maximum value. In order to understand the percentile values, lets consider the example of the variable 'balance'. For 'balance' 25% of the values are lower than 72. Median or the 50th percentile is 448. 75% of the values are either equal to or less than 1428; in other words, 25% or the values are greater than 1428.

```
bd.describe()
# display in pdf will be truncated on the right hand side
```

	age	balance	day	duration	campaign	pdays	previ
count	45211.000000	45211.000000	45211.000000	45211.000000	45211.000000	45211.000000	4521
mean	40.936210	1362.272058	15.806419	258.163080	2.763841	40.197828	0.580
std	10.618762	3044.765829	8.322476	257.527812	3.098021	100.128746	2.303
min	18.000000	-8019.000000	1.000000	0.000000	1.000000	-1.000000	0.000
25%	33.000000	72.000000	8.000000	103.000000	1.000000	-1.000000	0.000
50%	39.000000	448.000000	16.000000	180.000000	2.000000	-1.000000	0.000
75%	48.000000	1428.000000	21.000000	319.000000	3.000000	-1.000000	0.000
max	95.000000	102127.000000	31.000000	4918.000000	63.000000	871.000000	275.0

Another useful function that can be applied on the entire data is nunique(). It returns the number of unique values taken by different variables.

Numeric data

```
bd.nunique()
```

```
age            77
job            12
marital         3
education       4
default         2
balance      7168
housing         2
loan            2
contact         3
day            31
month          12
duration     1573
campaign       48
pdays         559
previous       41
poutcome        4
y               2
dtype: int64
```

We can observe that the variables having the 'object' type have fewer values and variables which are 'numeric' have higher number of unique values.

The describe() function can be used with individual columns also. For numeric variables, it gives the 8 summary statistics for that column only.

```
bd['age'].describe()
```

```
count    45211.000000
mean        40.936210
std         10.618762
min         18.000000
25%         33.000000
50%         39.000000
75%         48.000000
max         95.000000
Name: age, dtype: float64
```

When the describe() function is used with a categorical column, it gives the total number of values in that column, total number of unique values, the most frequent value ('blue-collar') as well as the frequency of the most frequent value.

```
bd['job'].describe()
```

```
count           45211
unique             12
top       blue-collar
freq             9732
Name: job, dtype: object
```

Note: When we use the describe() function on the entire dataset, by default, it returns the summary statistics of numeric columns only.

There are tonnes of different summary statistics available.

Lets say we only wanted the mean or the median of the variable 'age'.

```
bd['age'].mean(), bd['age'].median()
```

> (40.93621021432837, 39.0)

Apart from the summary statistics provided by the describe function, there are many other statistics available as shown below:

Function	Description
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
abs	Absolute Value
prod	Product of values
std	Unbiased standard deviation
var	Unbiased variance
sem	Unbiased standard error of the mean
skew	Unbiased skewness (3rd moment)
kurt	Unbiased kurtosis (4th moment)
quantile	Sample quantile (value at %)
cumsum	Cumulative sum
cumprod	Cumulative product
cummax	Cumulative maximum
cummin	Cumulative minimum

Till now we have primarily discussed how to summarize numeric data.

Categorical data

Now starting with categorical data, we would want to look at frequency counts. We use the value_count() function for get the frequency counts of each unique element present in the column.

```
1 bd['job'].value_counts()
```

```
blue-collar     9732
management      9458
technician      7597
admin.          5171
services        4154
retired         2264
self-employed   1579
entrepreneur    1487
unemployed      1303
housemaid       1240
student          938
unknown          288
Name: job, dtype: int64
```

By default, the outcome of the value_counts() function is in descending order. The element 'blue-collar' with the highest count is displayed on the top and that with the lowest count 'unknown' is displayed at the bottom.

We should be aware of the format of the output. Lets save the outcome of the above code in a variable 'k'.

```
k = bd['job'].value_counts()
```

The outcome stored in 'k' has two attributes. One is values i.e. the raw frequencies and the other is 'index' i.e. the categories to which the frequencies belong.

```
k.values
```

```
array([9732, 9458, 7597, 5171, 4154, 2264, 1579, 1487, 1303, 1240,  938,
       288], dtype=int64)
```

```
k.index
```

```
Index(['blue-collar', 'management', 'technician', 'admin.', 'services',
       'retired', 'self-employed', 'entrepreneur', 'unemployed', 'housemaid',
       'student', 'unknown'],
      dtype='object')
```

As shown, values contain raw frequencies and index contains the corresponding categories. e.g. 'blue-collar' job has 9732 counts.

Lets say, you are asked to get the category with minimum count, you can directly get it with the following code:

```
k.index[-1]
```

```
'unknown'
```

We observe that the 'unknown' category has the least count.

We can get the category with the second highest count as well as the highest count as follows:

```
k.index[-2] # returns category with the second lowest count
```

```
'student'
```

```
k.index[0] # returns category with the highest count
```

```
'blue-collar'
```

Now if someone asks us for category names with frequencies higher than 1500. We can write the following code to get the same:

```
k.index[k.values>1500]
```

```
Index(['blue-collar', 'management', 'technician', 'admin.', 'services',
       'retired', 'self-employed'],
      dtype='object')
```

Even if we write the condition on k itself, by default it means that the condition is applied in the values.

```
k.index[k>1500]
```

```
Index(['blue-collar', 'management', 'technician', 'admin.', 'services',
 'retired', 'self-employed'],
dtype='object')
```

The next kind of frequency table that we are interested in when working with categorical variables is cross-tabulation i.e. frequency of two categorical variables taken together. e.g. lets consider the cross-tabulation of two categorical variables 'default' and 'housing'.

```
pd.crosstab(bd['default'],bd['housing'])
```

housing	no	yes
default		
no	19701	24695
yes	380	435

In the above frequency table, we observe that there are 24695 observation where the value for 'housing' is 'yes' and 'default' is 'no'. This is a huge chunk of the population. There is a smaller chunk of about 435 observations where housing is 'yes' and default is 'yes' as well. Within the observations where default is 'yes', 'housing' is 'yes' for a higher number of observations i.e. 435 as compared to where housing is 'no' i.e. 380.

Now, lets say that we want to look at the unique elements as well as the frequency counts of all categorical variables in the dataset 'bd'.

```
bd.select_dtypes('object').columns
```

```
Index(['job', 'marital', 'education', 'default', 'housing', 'loan', 'contact',
 'month', 'poutcome', 'y'],
dtype='object')
```

The code above will give us all the column names which are stored as categorical datatypes in the 'bd' dataframe. We can then run a for loop of top of these columns to get whichever summary statistic we need for the categorical columns.

```
cat_var = bd.select_dtypes('object').columns
for col in cat_var:
    print(bd[col].value_counts())
    print('~~~~~')
```

```
blue-collar      9732
management       9458
technician       7597
admin.           5171
services         4154
retired          2264
self-employed    1579
entrepreneur     1487
unemployed       1303
housemaid        1240
student           938
unknown           288
Name: job, dtype: int64
~~~~~
married     27214
single      12790
divorced     5207
Name: marital, dtype: int64
~~~~~
secondary    23202
```

```
tertiary     13301
primary       6851
unknown       1857
Name: education, dtype: int64
~~~~~
no     44396
yes      815
Name: default, dtype: int64
~~~~~
yes    25130
no     20081
Name: housing, dtype: int64
~~~~~
no     37967
yes     7244
Name: loan, dtype: int64
~~~~~
cellular     29285
unknown      13020
telephone     2906
Name: contact, dtype: int64
~~~~~
may    13766
jul     6895
aug     6247
jun     5341
nov     3970
apr     2932
feb     2649
jan     1403
oct      738
sep      579
mar      477
dec      214
Name: month, dtype: int64
~~~~~
unknown    36959
failure     4901
other       1840
success     1511
Name: poutcome, dtype: int64
~~~~~
no     39922
yes     5289
Name: y, dtype: int64
~~~~~
```

Many times we do not only want the summary statistics of numeric or categorical variables individually; we may want a summary of numeric variables within the categories coming from a categorical variable. e.g. lets say we want the average age of the people who are defaulting their loan as opposed to people who are not defaulting. This is known as group wise summary.

```
bd.groupby(['default'])['age'].mean()
```

```
default
no    40.961934
yes   39.534969
Name: age, dtype: float64
```

The result above tells us that the defaulters have a slightly lower average age as compared to non-defaulters.

Looking at median will give us a better idea in case we have many outliers. We notice that the difference is not much.

```
bd.groupby(['default'])['age'].median()
```

```
default
no    39
yes   38
Name: age, dtype: int64
```

We can group by multiple variables as well. There is no limit on the number and type of variables we can group by. But generally, we group by categorical variables only.

Also, it is not necessary of give the name of the column for which we want the summary statistic. e.g. in the code above, we wanted the median of the column 'age'. It is not necessary to specify the column 'age'. When we do not specify the column age, then we get a median of all the numeric columns grouped by the variable 'default'.

```
bd.groupby(['default']).median()
```

	age	balance	day	duration	campaign	pdays	previous
default							
no	39	468	16	180	2	-1	0
yes	38	-7	17	172	2	-1	0

We can also group by multiple variables as follows:

```
bd.groupby(['default','loan']).median()
```

		age	balance	day	duration	campaign	pdays	previous
default	**loan**							
no	**no**	39.0	509.0	16.0	181.0	2.0	-1.0	0.0
	yes	39.0	284.0	17.0	175.0	2.0	-1.0	0.0
yes	**no**	38.0	-3.5	17.0	178.5	2.0	-1.0	0.0
	yes	39.0	-21.0	18.0	163.0	2.0	-1.0	0.0

Each row in the result above gives the 4 categories defined by the two categorical variables we have grouped by and each column give the median value for all the numerical variables for each group.

In short, when we do not give a variable to compute the summary statistic e.g. median, we get all the columns where median can be computed.

Now, lets say we do not want to find the median for all columns, but only for 'day' and 'balance' columns. We can do that as follows:

```
bd.groupby(['housing','default'])['balance','day'].median()
```

		balance	day
housing	**default**		
no	**no**	531	17
	yes	0	18
yes	**no**	425	15
	yes	-137	15

Visual summary of data

Here we will understand how to summarize data visually. We will use the 'seaborn' library for this. Its a high level package built using matplotlib which is the base python visualization library.

If we were to do the visualizations using matplotlib instead of seaborn, we would need to write a lot more code. Seaborn has functions which wrap up this code making simpler.

```
import seaborn as sns
%matplotlib inline
```

Note: `%matplotlib inline` is required only when we use the Jupyter notebook so that visualizations appear within the notebook itself. Other editors like Spyder or PyCharm do not need this line of code as part of our script.

What we will cover is primarily to visualize our data quickly which will help us build our machine learning models.

We will learn to visualize the following:

1. Single numeric variables - density plots, box plots
2. Numeric numeric variables - scatter plot, pairwise density plots
3. Faceting of the data (visualizing chunks of data broken on the basis of categorical variables)
4. Categorical variables - frequency bar plots
5. Faceting of categorical variables
6. Heatmaps

```
import pandas as pd
import numpy as np
```

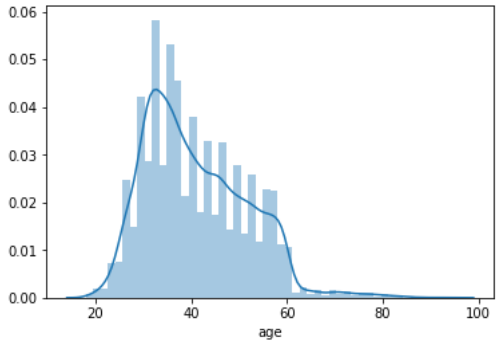
```
file=r'/Users/anjal/Dropbox/PDS V3/Data/bank-full.csv'
bd=pd.read_csv(file,delimiter=';')
```

1. Visualizing single numeric variable

Density plots

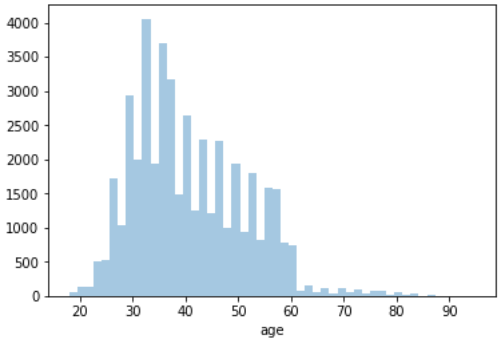
Lets start with density plots for a single numeric variable. We use the distplot() function from the seaborn library to get the density plot. The first argument will be the variable for which we want to make the density plot.

```
sns.distplot(bd['age'])
```



By default, the distplot() function gives a histogram along with the density plot. In case we do not want the density plot, we can set the argument 'kde' (short for kernel density) to False.

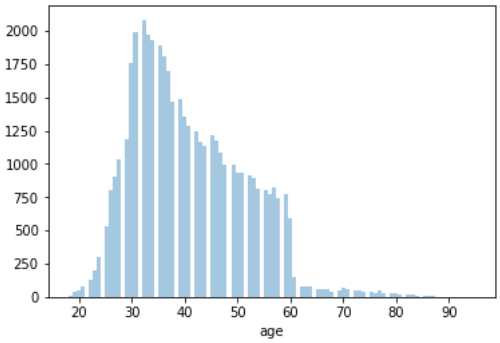
```
sns.distplot(bd['age'], kde=False)
```



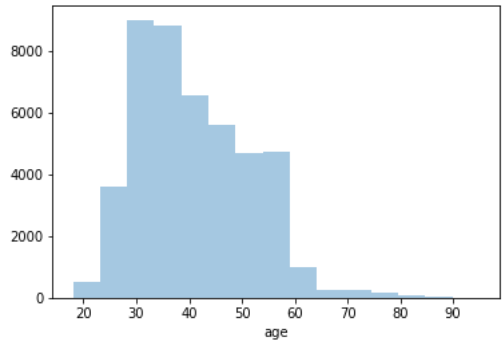
Setting the 'kde' argument to False will not show us the density curve, but will only show the histogram.

In order to build a histogram, continuous data is split into intervals called bins. The argument 'bins' lets us set the number of bins which in turn affects the width of each bin. This argument has some default value, however we can increase the number of bins by changing the 'bin' argument.

```
sns.distplot(bd['age'], kde=False, bins=100)
```




```
sns.distplot(bd['age'], kde=False, bins=15)
```

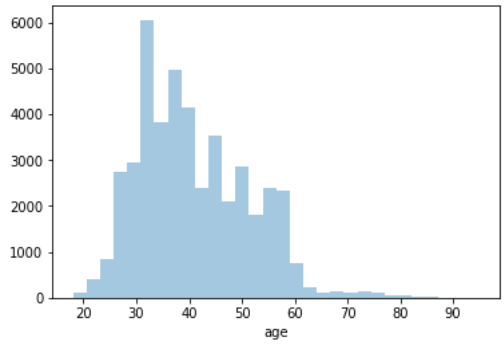


Notice the difference in the width of the bins when the argument 'bins' has different values. With 'bins' argument having the value 15, the bins are much wider as compared to when the 'bins' have the value 100.

How do we decide the number of bins, what would be a good choice? First consider why we need a histogram. Using a histogram we can get a fair idea where most of the values lie. e.g. considering the variable 'age', a histogram tells us how people in the data are distributed across different values of 'age' variable. Looking at the histogram, we get a fair idea that most of the people in our data lie in 30 to 40 age range. If we look a bit further, we can also say that the data primarily lies between 25 to about 58 years age range. Beyond the age of 58, the density falls. We might be looking at typical working age population.

Coming back to how do we decide the number of bins. Now, we can see that most of the people are between 30 to 40 years. Here, if we want to dig deeper to see how data is distributed within this range we increase the number of bins. In other words, when we want to go finer, we increase the number of bins.

```
1 sns.distplot(bd['age'], kde=False, bins=30)
```

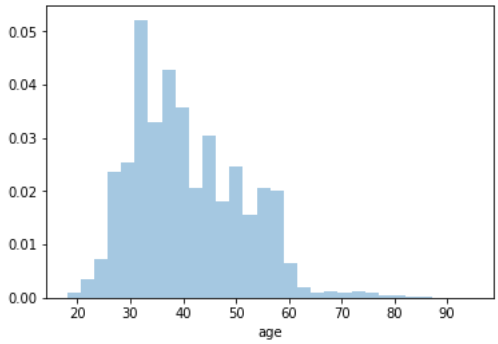


We can see here that between 30 to 40, the people in their early 30's are much more dominant as compared to the people whose age is closer to 40. One thing to be kept in mind when increasing the number of bins is that if the number of data points are very low, for example, if we have only 100 data points then it does not make sense to create 50 bins because the frequency bars that we see will not give us a very general picture.

In short, higher number of bins will give us a finer picture of how the data is behaving in terms of density across the value ranges. But with very few data points, a higher number of bins may give us a picture which may not be generalizable.

We can see that the y axis has frequencies. Sometimes it is much easier to look at frequency percentages. We get frequency percentages by setting the 'norm_hist' argument to True. It basically normalizes the histograms.

```
1 sns.distplot(bd['age'], kde=False, bins=30, norm_hist=True)
```

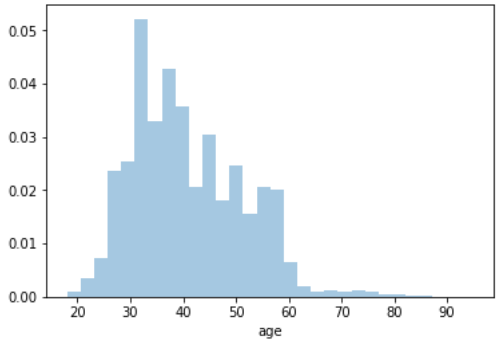


We can see that about 5% of the population lies between the age range of 31 to about 33.

Note: If we want to get more specific, we need to move towards numeric summary of data.

How do we save this graph as an image?

```
myplot = sns.distplot(bd['age'], kde=False, bins=30, norm_hist=True)
```



The code above saves the visualization in myplot.

```
myimage = myplot.get_figure()
```

myimage is something that we can save.

```
myimage.savefig("output.png")
```

The moment we do this, the 'output.png' file will appear wherever this script is. We can get this path using the pwd() function. My 'output.png' file is present in the following path:

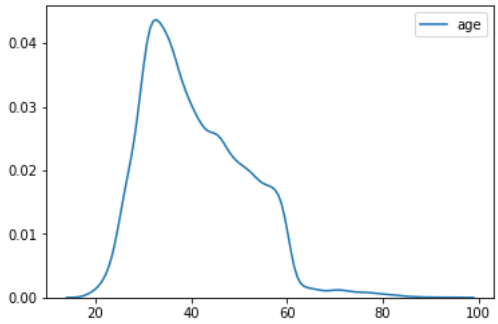
```
pwd()
```

> 'C:\Users\anjal\Dropbox\PDS V3\2.Data_Prep'

The above method of saving images will work with all kinds of plots.

There is a function kdeplot() in seaborn that is used for generating only the density plot. It will only give us the density plot without the histogram.

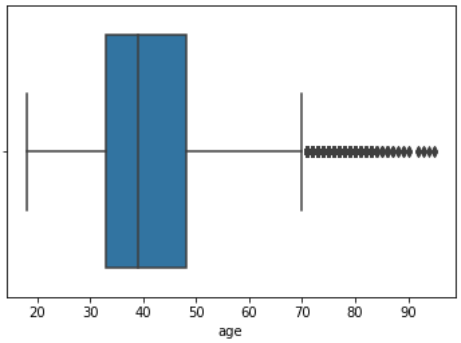
```
sns.kdeplot(bd['age'])
```



We observe that the different plots have their own functions and we simply need to pass the column which we wish to visualize.

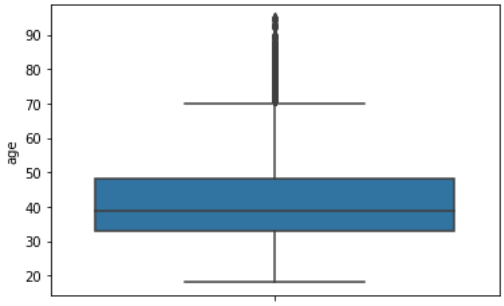
Now, let us see how to visualize the 'age' column using a boxplot.

```
sns.boxplot(bd['age'])
```



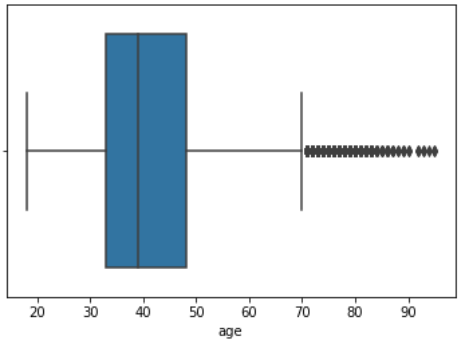
We can also get the above plot with the following code:

```
sns.boxplot(y='age', data=bd)
```



We get a vertical boxplot since we mentioned y as 'age'. We can also get the horizontal boxplot if we specify x as 'age'.

```
sns.boxplot(x='age', data=bd)
```

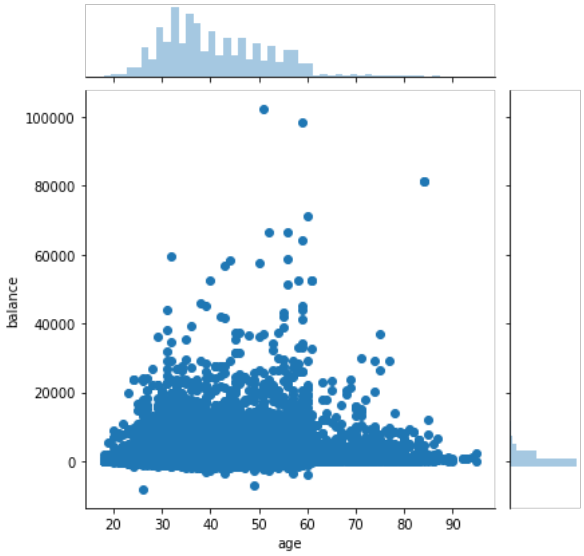


We can see that there are no extreme values on lower side of the age; however, there are a lot of extreme values on the higher side of age.

Visualizing numeric-numeric variables

We can use scatterplots to visualize two numeric columns together. The function which helps us do this is jointplot() from the seaborn library.

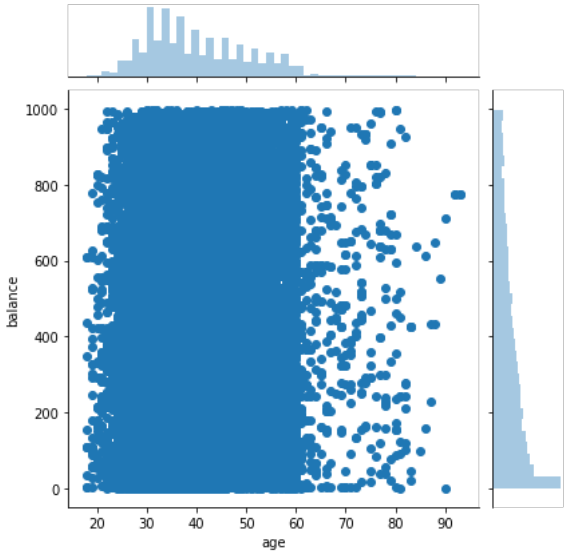
```
sns.jointplot(x='age', y='balance', data=bd)
```



The jointplot not only gives us the scatterplot but also gives the density plots along the axis.

We can do a lot more things with the jointplot() function. We observe that the variable 'balance' takes a value on a very long range but most of the values are concentrated on a very narrow range. Hence we will plot the data only for those observations for which 'balance' column has values ranging from 0 to 1000.

```
sns.jointplot(x="age", y="balance", data=bd.loc[((bd['balance']>0) & (bd['balance']
<1000)),:])
```

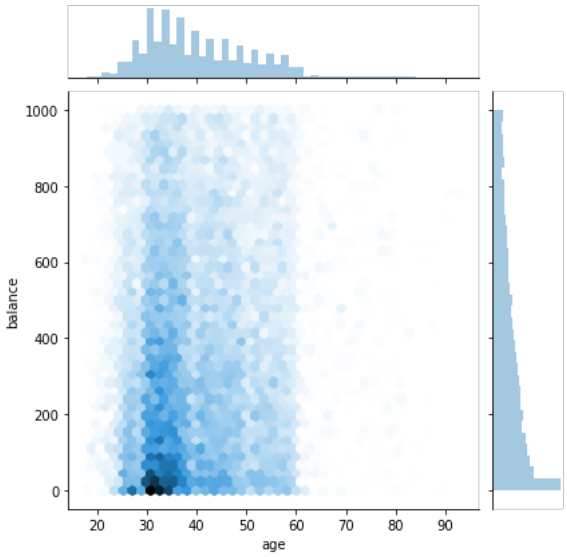


Note: Putting conditions is not a requirement for us to visualize the data. Since we see that most of the data lies in a smaller range and because of the few extreme values we may get a distorted plot.

Using the above code we have no way of figuring out if individual points are overlapping each other.

Setting the argument 'kind' as 'hex' not only shows us the observations but also helps us in knowing how many observations are overlapping at a point by observing the shade of each point. The darker the points, more the number of observations present there.

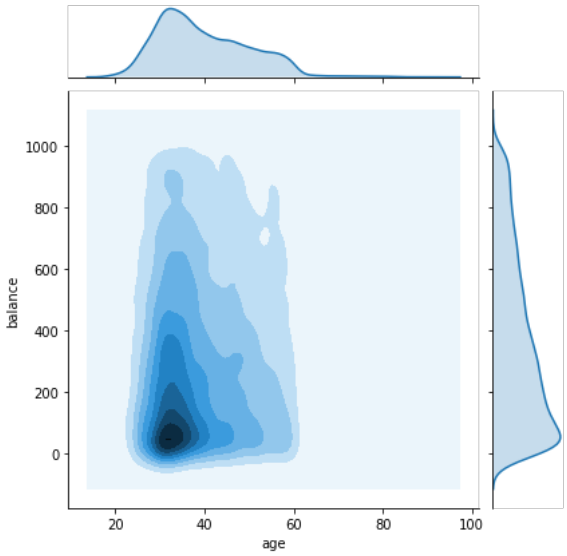
```
sns.jointplot(x="age", y="balance", data=bd.loc[((bd['balance']>0) & (bd['balance']<1000)),:], kind='hex')
```



We can observe that most of the observations lie between the age of 30 and 40 and lot of observations lie between the balance of 0 to 400. As we move away, the shade keeps getting lighter indicating that the number of observations reduce or the density decreases.

These hex plots are a combination of scatterplots and pure density plots. If we want pure density plots, we need to change the 'kind' argument to 'kde'.

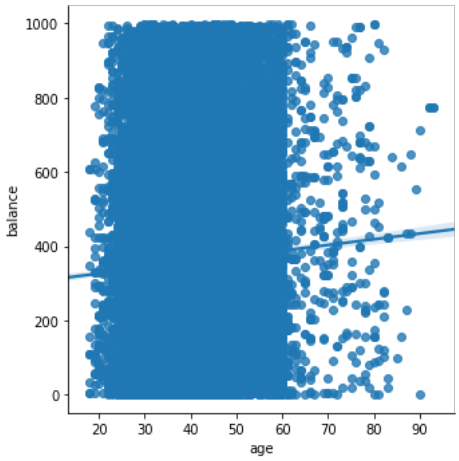
```
sns.jointplot(x="age", y="balance", data=bd.loc[((bd['balance']>0) & (bd['balance']<1000)),:], kind='kde')
```



The darker shade indicates that most of the data lies there and as we move away the density of the data dissipates.

Next we will see the lmplot() function.

```
sns.lmplot(x='age', y='balance', data=bd.loc[((bd['balance']>0) & (bd['balance']<1000)),:])
```

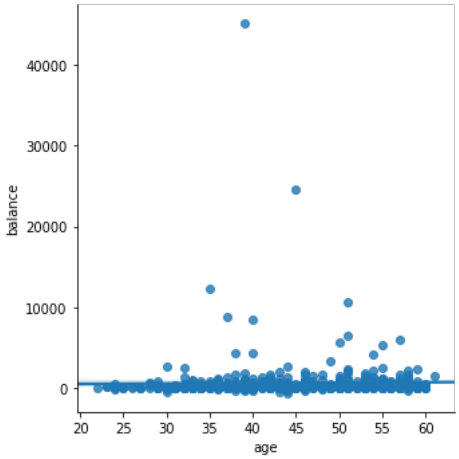


We observe that lmplot is just like scatter plot, but it fits a line through the data by default. (lmplot - linear model plot)

We can update lmplot to fit higher order polynomials which gives a sense if there exists a non-linear relationship between the data.

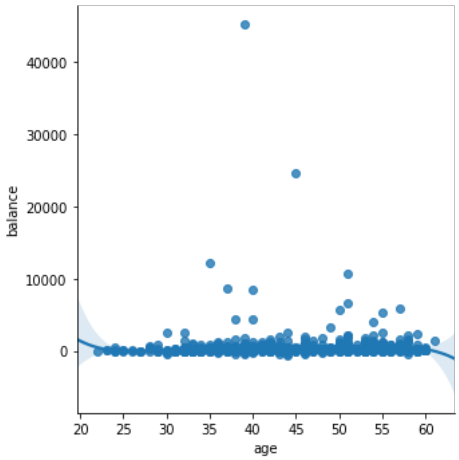
Since the data in the plot above is overlapping a lot, let us consider the first 500 observations only. We can see that a line fits through these points.

```
sns.lmplot(x='age', y='balance', data=bd.iloc[:500,:])
```



If we update the code above and add an argument 'order=6' we can see that the function has tried to fit a curve through the data. Since it still mostly looks like a line, so maybe there is a linear trend. Also, as the line is horizontal to the x axis, there is not much correlation between the two variables plotted.

```
sns.lmplot(x='age', y='balance', data=bd.iloc[:500,:], order=6)
```

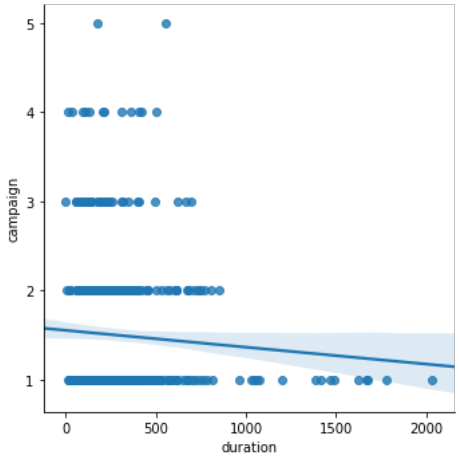


3. Faceting the data

As of now, we are looking at the age and balance relationship across the entire data.

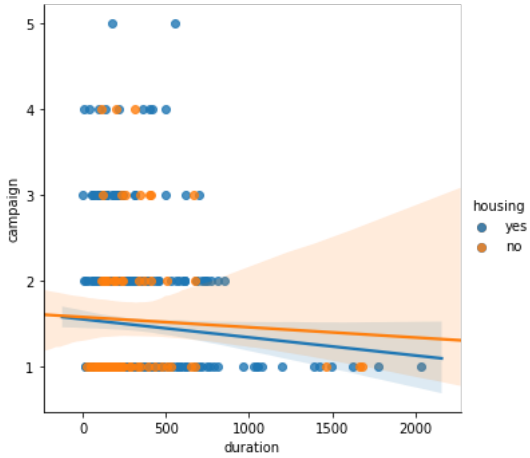
Let us see the relationship between 'duration' and 'campaign' variables.

```
1 sns.lmplot(x='duration', y='campaign', data=bd.iloc[:500,:])
```



Now we want to see how will the relationship between 'duration' and 'campaign' behave for different values of 'housing'. We can observe this by coloring the datapoints for different values of 'housing' using the 'hue' argument. 'housing' variable takes two values: yes and no.

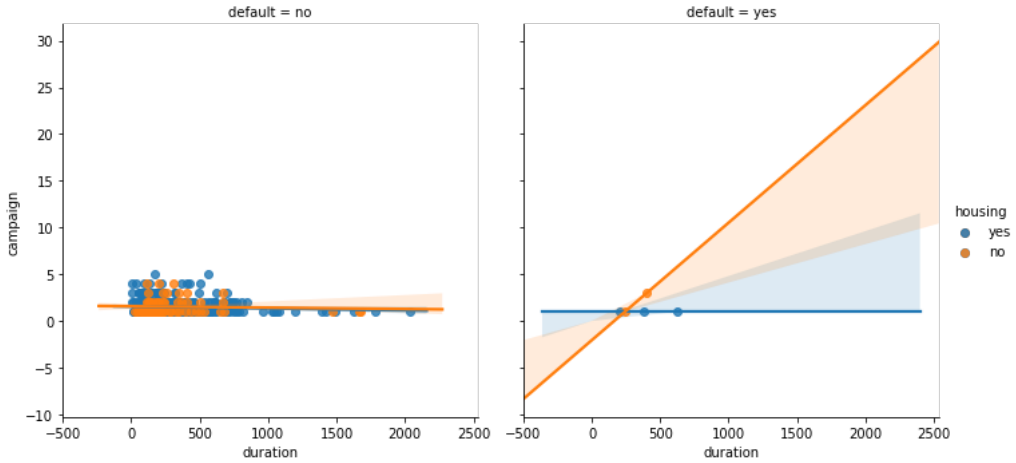
```
1 sns.lmplot(x='duration', y='campaign', data=bd.iloc[:500,:], hue='housing')
```



We can see two different fitted lines. The orange one corresponds to 'housing' equal to no and the blue one corresponding to 'housing' equal to yes.

Now if we wish to divide our data further on the basis of 'default' column, we can consider using the 'col' argument. 'default' argument takes two values: yes and no.

```
sns.lmplot(x='duration', y='campaign', data=bd.iloc[:500,:], hue='housing',
col='default')
```

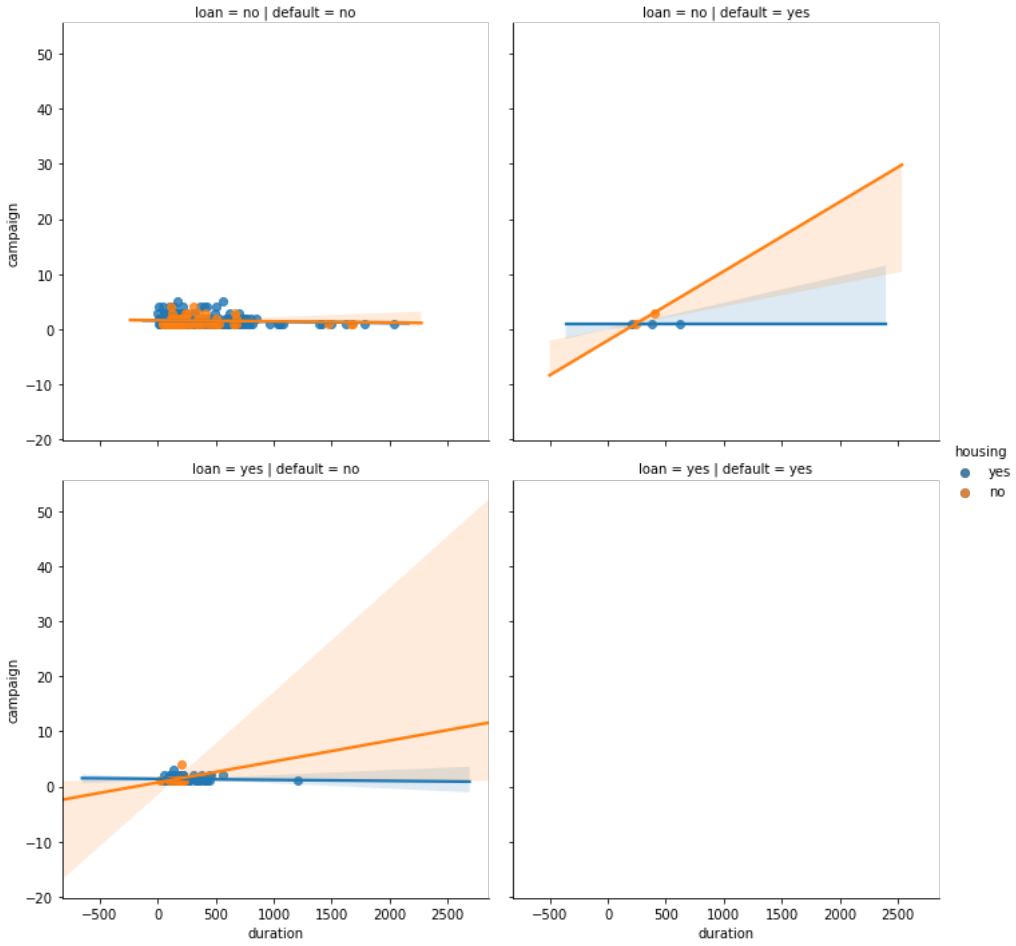


Now we have 4 parts of the data. Two are given by the color of the points and two more are given by separate columns. The first column refers to 'default' being no and the second column refers to 'default' being yes.

Observe that there are very few points when 'default' is equal to yes; hence we cannot trust the relationship as the data is very less.

Next, if we wanted to check how does the relationship between the two variables change when we are looking at different categories for loan. 'loan' also has two values: yes and no.

```
sns.lmplot(x='duration', y='campaign', data=bd.iloc[:500,:], hue='housing',
col='default', row='loan')
```



We observe, that within the group where 'default' is no; the relationship between the two variables 'campaign' and 'duration' is different when 'loan' is yes as compared to when 'loan' is no. Also, majority of the data points are present where 'loan' and 'default' both are no. There are no data points where both 'loan' and 'default' is yes. Keep in mind that we are looking at the first 500 observations only. We may get some data points here if we look at higher number of observations.

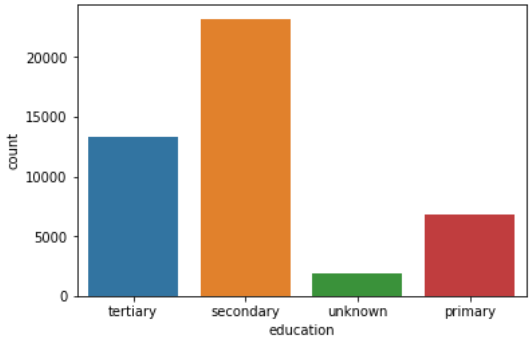
This how we can facet the data observing whether after breaking the data does the relationship between two variables change.

4. Visualizing categorical data

Lets start by making simple frequency bar charts using count plots.

We want to know how different education groups are present in the data.

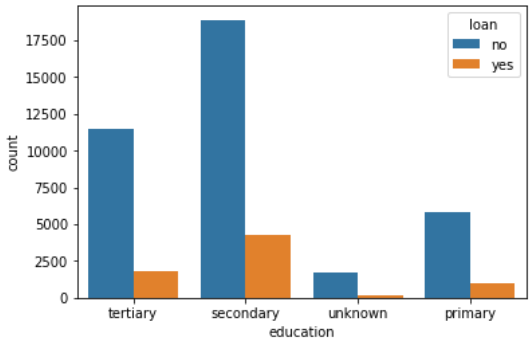
```
1 sns.countplot(x = 'education', data = bd)
```



We observe that the 'secondary' education group is very frequent. There is a small chunk where the level of education is 'unknown'. The 'tertiary' education group has the second highest count followed by 'primary'.

We have options to use faceting here as well. We can use the same syntax we used earlier. Lets start by adding 'hue' as 'loan'.

```
sns.countplot(x='education', data=bd, hue='loan')
```



We observe that each education level is now broken into two parts according to the value of 'loan'.

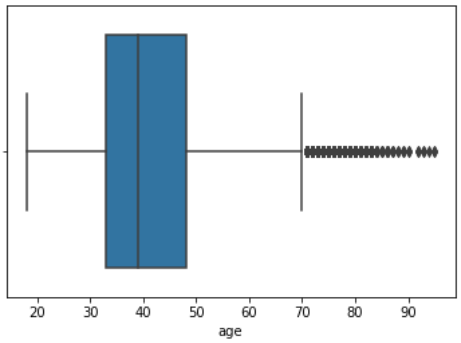
This is how we can facet using the same options for categorical data.

When we want to see how 'age' behaves across different levels of education, we can use boxplots().

When we make a boxplot only with the 'age' variable we get the following plot, indicating that the data primarily lies between age 20 and 70 with a few outlying values and the data overall is positively skewed.

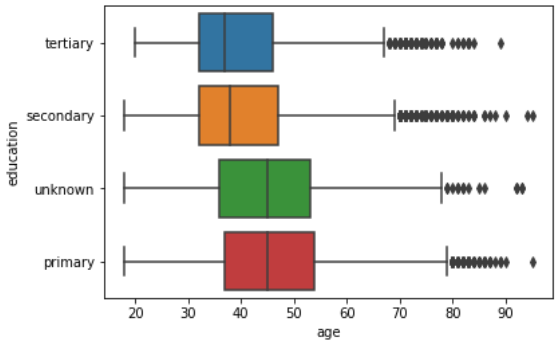
5. Faceting of categorical variables

```
sns.boxplot(x='age', data=bd)
```



Now, we want to look how the variable 'age' behaves across different levels of 'education'.

```
sns.boxplot(x='age', y='education', data=bd)
```



We observe that 'primary' education have overall higher median range. The behavior of the data points under 'unknown' are similar to those under 'primary' education indicating maybe that the 'unknown' ones may have primary education background.

We also observe that people having 'tertiary' education belong to a lower age group. We can infer that older people could make do with lesser education but the current generation needs higher levels of education to get by. This could be one of the inferences.

6. Heatmaps

Heatmaps are two dimensional representation of data in which values are represented using colors. It uses a warm-to-cool color spectrum to describe the values visually.

In order to understand heatmaps, lets start with generating a random array.

```
x = np.random.random(size=(20,20))
```

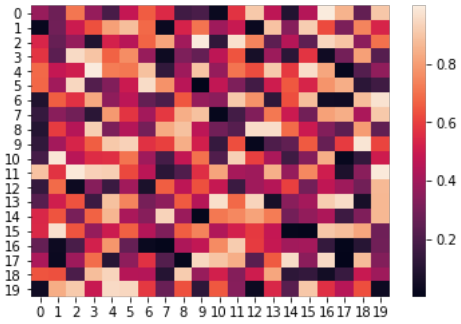
```
x[:3]
```

```
array([[0.37666211, 0.28033254, 0.71360714, 0.37308941, 0.19125123,
        0.48272533, 0.65697561, 0.54930338, 0.17009858, 0.19373981,
        0.02635841, 0.57243852, 0.90673713, 0.4668148 , 0.09514138,
        0.44202699, 0.99756982, 0.85043029, 0.25563275, 0.90301468],
       [0.02407724, 0.2872011 , 0.5026282 , 0.63615388, 0.80489701,
        0.88052132, 0.68300948, 0.0335487 , 0.52819662, 0.70863449,
        0.39304293, 0.48927019, 0.02993629, 0.89080078, 0.33833946,
        0.91885461, 0.62108977, 0.31585145, 0.74250102, 0.5337096 ],
       [0.53687923, 0.26465329, 0.37573166, 0.09014275, 0.52514711,
        0.45017881, 0.67338532, 0.81795034, 0.39931306, 0.99530252,
        0.12930078, 0.96921   , 0.74296808, 0.24080314, 0.44930359,
        0.24761788, 0.94031158, 0.89358538, 0.35454129, 0.7008932 ]])
```

Looking at the data above, it is difficult for us to determine what kind of values it has.

If we pass this data to the heatmap() function, it will be displayed as below:

```
sns.heatmap(x)
```



When we observe a heatmap, wherever the color of the boxes are light, the values are closer to 1 and as the boxes get darker, those are the values closer to 0. Looking at the visualization above, we get an idea that more or less the values are quite random. There does not seems to be any dominance of lighter or darker boxes.

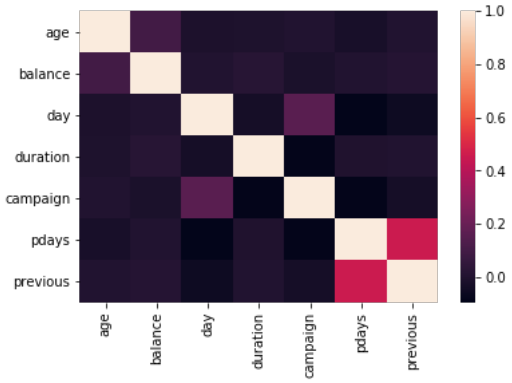
Now, since we understand the use of colors in heatmaps, lets get back to understanding how does it help with understanding our 'bd' dataset. Lets say we look at the correlations in the data 'bd'.

```
bd.corr()
```

	age	balance	day	duration	campaign	pdays	previous
age	1.000000	0.097783	-0.009120	-0.004648	0.004760	-0.023758	0.001288
balance	0.097783	1.000000	0.004503	0.021560	-0.014578	0.003435	0.016674
day	-0.009120	0.004503	1.000000	-0.030206	0.162490	-0.093044	-0.051710
duration	-0.004648	0.021560	-0.030206	1.000000	-0.084570	-0.001565	0.001203
campaign	0.004760	-0.014578	0.162490	-0.084570	1.000000	-0.088628	-0.032855
pdays	-0.023758	0.003435	-0.093044	-0.001565	-0.088628	1.000000	0.454820
previous	0.001288	0.016674	-0.051710	0.001203	-0.032855	0.454820	1.000000

Normally the correlation tables can be quite huge, can have 50 variables in the data too. Looking at this table, it is very difficult to manually check if there exists correlation between the variables. We can manually check if any of the values in the table above are close to 1 or -1 which indicates high correlation or we can simply pass the above table to a heatmap.

```
1 sns.heatmap(bd.corr())
```



The visualization above shows that wherever the boxes are very light i.e. near +1, there is high correlation and wherever the boxes are too dark i.e. near 0, the correlation is low. The diagonal will be the lightest because each variable has maximum correlation with itself. But for the rest of the data, we observe that there is not much correlation. However, there seems to be some positive correlation between 'previous' and 'pdays'. The advantage of using heatmaps is that we do not have to go through the correlation table to manually check if correlation is present or not. We can visually understand the same through the heatmap shown above.

Chapter 3 : Introduction to Machine Learning

We'll start our discussion with one thing that people tend to forget over time. Whatever we are learning is about solving business problems .Lets start with; `where do we start after we are given a business problem to work on; in context of machine learning`.

Converting Business Problems to Data Problems

At the end of the day; whatever we are doing here is about money. To put that more appropriately , we are developing these techniques to solve business problems. Real business problems do not come conveniently all dressed up as a data problem in general. For example , consider this :

- A bank is making too many losses because of defaulters on retail loans

Its a genuine business problem but it isn't a data problem yet on the face of it. So, what is a data problem then. A data problem is a business problem expressed in terms of the data [potentially] available in the business process pipeline.

It majorly has two components:

- Response/Goal/Outcome
- Set of factor/features data which affects our goal/response/outcome

Lets look at the loan default problem and find these components.

- Outcome is loan default, which we would like to predict when considering giving loan to a prospect.
- What factors could help us in doing that? Banks collect a lot of information on a loan application such as financial data, personal information. In addition to that they also make queries regarding credit history to various agencies. We could use all these features/information to predict whether a customer is going to default on their loan or not and then reconsider our decision of granting loans depending on the result.

Here are few more business problems which you can try converting to data problems :

- A marketing campaign is causing spam complaints from the existing customers
- Hospitals can not afford to have additional staff all year round
  which are needed only when patient intakes become higher than a certain amount
- An eCommerce company wants to know how it should plan for the budget on cloud servers
- How many different election campaign strategies should a party opt for
- What kind of toys should Lego launch in India

Three kinds of Problems

If you went through the business problems mentioned above, you'd have realized there are mainly three kind of problems which can then be clubbed into two categories :

1. Supervised
2. Unsupervised

Supervised problems are the problems which have explicit outcomes. Such as default on loan, Required Number of Staff, Server Load etc. Within these , you can see separate kinds.

1. Regression
2. Classification

Regression problems are those where outcome is a continuous numeric value e.g. Sales , Rainfall , Server Load (values over a range with technically infinite unique values possible as outcome and have an ordinal relationship [e.g. 100 is twice as much as to 50]).

Classification problem on the other hand have their outcome as categories [e.g.: good/bad/worse; yes/no ; 1/0 etc], with limited number of defined outcomes .

Unsupervised problems are those where there is no explicit measurable outcome associated with the problem. You just need to find general pattern hidden in the measured factors. Finding different customer segments or electoral segments or finding latent factors in the data comes under such problems.

Now categories of problems can be formally grouped like this:

1. Supervised
    1. Regression
    2. Classification
2. Unsupervised

Our focus in this module will be over **Supervised** problems with an existing outcome which we are trying to predict in context of a regression or a classification problem

Data Problem to Mathematical Problem

Lets discuss key takeaways from the discussion above; which will help us in converting a data problem (of supervised kind) to its mathematical representation.

- We want to make use of historical data to extract pattern so that we can build a predictive model for future/new data
- We want our solution to be as accurate as possible

Now, what do we mean by pattern ? In mathematical terms; we are looking for a function which takes input (the values of factors affecting my response/outcome) and outputs the value of outcome (which we call predictions)

Regression Problem

We'll denote our prediction for i^{th} observation as $\hat{y_i}$ and real value of the outcome in the historical data given to us as y_i . And this function that we talk about is denoted by `f` . Inputs collectively are denoted as X_i

$$\hat{y_i} \ = \ f(X_i)$$

It isn't really possible to have a function which will make perfect predictions [in fact it is possible but not good to have a function which makes perfect prediction, its called over-fitting ; we'll keep that discussion for later] .

Our predictions are going to have errors . We can calculate those errors easily by comparing our predictions with real outcomes .

$$e_i \ = \ y_i \ - \ \hat{y_i}$$

We would want this error to be as small as possible across all observations. One way to represent the error `across all observation` will be average error for the entire data . However simple average is going to be meaning less , because errors for different observations might have different signs; some negative some positive. Overall average error might as well be zero, but that doesn't mean individual errors don't exist. We need to come up with something for this error `across all observation` which doesn't consider sign of errors . There are couple of ideas that we can consider .

- Mean Absolute Error $\sum_i^{i=n} |e_i|$
- Mean Squared Error $\sum_i^{i=n} {e_i}^2$

These here are called **Cost Functions**, another popular name for the same is Loss Function . They are also mentioned as just Cost/Loss . In many places , these are considered as averages but simple sum [Sum of absolute errors or sum of squared errors] . It doesn't really matter because difference between them is division by a constant (number of observations); it doesn't affect our pattern extraction or function estimation for prediction.

Among the two that we mentioned here , Mean Squared Error is more popular in comparison to Mean Absolute Error due to its easy differentiability. How does that matter? It'll be clear once we discuss `Gradient Descent`.

Lets take a pause here and understand , what do these cost functions represent? These represent our business expectation of model being as accurate as possible in a mathematically quantifiable way .

We would want our prediction function $f(X_i)$ to be such that , for the given historical data, Mean Squared Error (or whatever other cost function you are considering) is as small as possible .

Classification Problem

It was pretty straight forward, as to what do we want to predict in Regression Problem. However it isn't that simple for classification problem as you might expect . Consider the following data on customer's Age and their response to a product campaign for an insurance policy.

Age	Outcome	Age	Outcome
20	NO	30	NO
20	NO	30	NO
20	NO	30	YES
20	NO	30	YES
20	YES	30	YES
25	NO	35	NO
25	NO	35	YES
25	NO	35	YES
25	YES	35	YES
25	YES	35	YES

If I asked you , what will be the outcome if somebody's age is 31, according to the data shown above . Your likely answer is `YES` as you see that majority of the people with increasing age have said `YES`.

If I further asked you whats the outcome for somebody with age 34, your response will again be `YES` . Now, whats the difference between these two cases . You can easily see that outcome is **more likely** to be `YES` when the age is higher . This tells us that , we are not really interested in absolute predictions `YES/NO`; but in the probability of the outcome being `YES/NO` for given inputs.

More formally we are interested in this : $P(y_i = YES|X_i)$

`YES/NO` at the end of the day are just labels. We'll consider them to be 1/0 to make our life mathematically easy; as will be evident in some time. Also to keep the notation concise , we'll use just p_i instead of $P(y_i = 1|X_i)$

Now that we have figure out that we want to predict p_i, Lets see how do we write that against our prediction function $f(X_i)$

range of p_i being a probability is between 0 to 1. However $f(X_i)$ theoretically can be anything between $-\infty$ to $+\infty$. it wont make sense to write $p_i = f(X_i)$

we need to apply some transformation on any one side to ensure that the ranges match. sigmoid or logit function is one such transformation which is popular in use.

$$p_i = \frac{1}{1 + e^{-f(X_i)}}$$

another format [just rearranged terms] for the same is

$$log(\frac{p_i}{1 - p_i}) = f(X_i)$$

Finally, we are clear about what we want to predict and how it is written against our prediction function . We can now start discussing what are our business expectation from this and how do we represent the same in form of a cost/loss function .

We'll ideally want the probability of outcome being 1 to 100% when outcome in reality is 1 , and probability of outcome being 1 to be 0% when outcome in reality is 0 . But, as usual , perfect solution is practically not possible. Closest compromise will be that p_i is as close to 1 as possible when outcome in reality is 1 and it is as close to 0 as possible when outcome in reality is 0.

$$\begin{aligned} p_i \uparrow \quad & \forall \quad y_i = 1 \\ p_i \downarrow \quad & \forall \quad y_i = 0 \end{aligned}$$

since p_i takes value in the range [0,1] , we can say that when p_i goes close to 0, it implies that $1 - p_i$ goes close to 1. So the above expression can be written as :

$$\begin{aligned} p_i \uparrow \quad & \forall \quad y_i = 1 \\ (1 - p_i) \uparrow \quad & \forall \quad y_i = 0 \end{aligned}$$

This is still at intuition level, we need to find a way to convert this idea into a mathematical expression which can be used as a cost function . Consider this expression :

$$L_i = {p_i}^{y_i} (1 - p_i)^{(1 - y_i)}$$

Expression L_i is known as likelihood because it gives you the probability of the real outcome that you get. This expression takes value p_i when $y_i = 1$ and takes value $(1 - p_i)$ when $y_i = 0$. Using the idea that we devised above, we can rewrite it like this :

$$\begin{aligned} L_i \uparrow \quad & \forall \quad y_i = 1 \\ L_i \uparrow \quad & \forall \quad y_i = 0 \end{aligned}$$

This implies that we want the likelihood to be as high as possible irrespective of what the outcome (1/0) is. Another way to express that is , that we want likelihood to align with whatever the real outcome is . As earlier we are not concerned with likelihood of a single observation, we are interested in it at over all level. Since likelihood is nothing but a probability , if we want to calculate likelihood of entire data, it'll be nothing but multiplication of all individual likelihoods.

$$L = \prod_{i=1}^{i=n} L_i$$

This is our cost function which we need to maximize . we want such an $f(X_i)$ for which L is as high possible for the given historical data. However this is not the form in which this is used . Since optimizing a quantity where individual terms are multiplied with each other is a very difficult problem to solve we'll instead use log(L) as our cost function , which makes individual terms getting summed up instead .

$$log(L) = \sum_{i=1}^{i=n} L_i$$

To make this a standard minimization problem , instead of maximizing log(L), minimization of -log(L) is considered

Finally , cost function for classification problem is -log(L) , also known as **negative log likelihood**

other forms of the same which you'll get to see :

$$-log(L) = -\sum_{i=1}^{i=n}[y_i log(p_i) + (1 - y_i) log(1 - p_i)]$$

this can also be written in terms of prediction function as follows :

$$-log(L) = -\sum_{i=1}^{i=n}[y_i f(X_i) - log(1 + e^{f(X_i)})]$$

Estimating $f(X_i)$

For this discussion we'll consider regression with $f(X_i)$ as a linear model , but the same idea will be applicable to any parametric model

Lets say we are trying to predict sales of a jewelry shop, considering gold price that day and temp that day as factors affecting sales . We can consider a linear model like this

$$sales_i = \beta_0 + \beta_1 * price_i + \beta_2 * temp_i$$

Here βs are some constants. How do we determine, what values of βs should we consider ?

As per the discussion that we have had about business expectation , we would desire βs for which the cost function is as low as possible for the given historical data given to us .

We can try out many different values of βs and select the ones for which our cost function is coming out to be as low as possible. One way to find good βs can be to simply compare the values of cost functions for each and pick the one with lowest cost function value.

Here are few steps of the process :

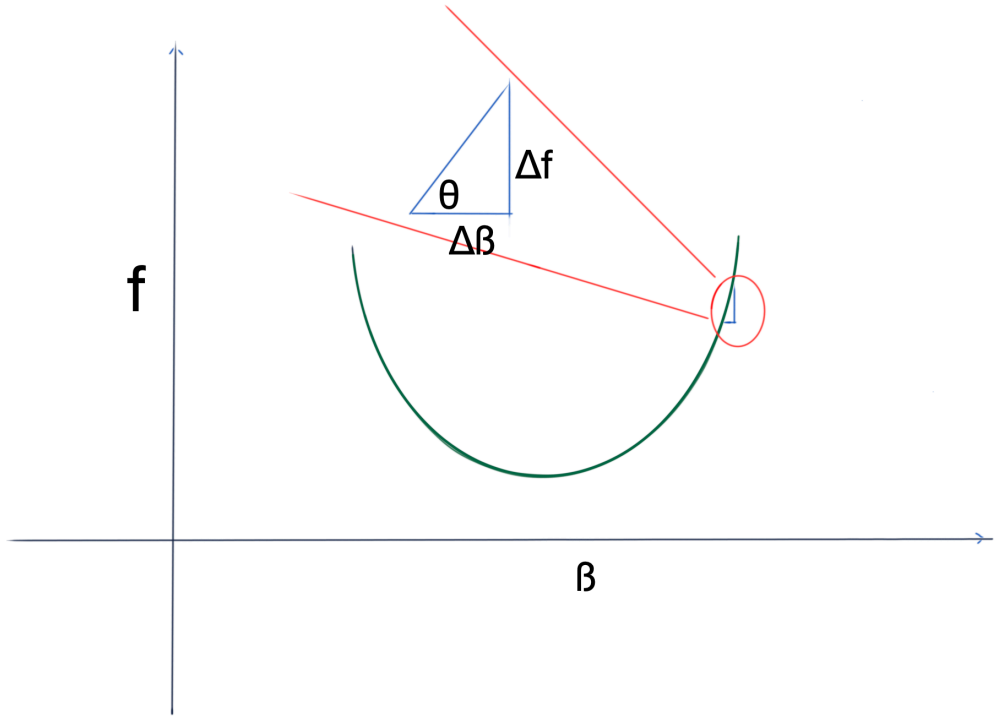
β_0	β_1	β_2	**Cost**	**Decision**
10	5	0	140	Start
-5	-2	4	160	Discard
1	3	7	150	Discard
4	7	02	135	Switch to theses Values
2	-2	10	130	Switch to theses Values
1	4	6	132	Discard
3	6	-5	141	Discard

We can keep on trying random values of βs like that and switching to new values whenever we encounter a combination which gives us a new low for the cost function.

And we can stop randomly trying new values of βs if we don't get any lower value for cost function for a long time .

This works in theory , given ; that we have infinite amount of time, resources and most important; patience!. We would want to have another method, which enables us to change our βs in a such a way that cost function always goes down [instead of changing randomly and discarding most of them] .

Gradient Descent



Here in this image we can see that f is function with parameter β . If we change β by a small amount $\Delta\beta$, function changes by Δf . As long as we ensure that $\Delta\beta$ is small , we can assume the triangle shown with sides $\Delta\beta$ and Δf, is right angle triangle , and we can write :

$$tan(\theta) = \frac{\Delta f}{\Delta \beta} \qquad (1)$$

For very small $\Delta\beta$ we can write $tan(\theta)$ in terms of gradient as follows:

$$tan(\theta) = \frac{\delta f}{\delta \beta} \qquad (2)$$

Equating this to (1) and after doing a little adjustment we get:

$$\Delta f = \frac{\delta f}{\delta \beta}\Delta\beta \qquad (3)$$

if there were multiple parameters involved , say β_1 and β_2

then changing those parameters will individually contribute to changing the function

$$\Delta f = \frac{\delta f}{\delta \beta_1}\Delta\beta_1 + \frac{\delta f}{\delta \beta_2}\Delta\beta_2 \qquad (4)$$

this can be written as dot product between two vectors , respectively know as gradient and change in parameters

$\nabla f = (\frac{\delta f}{\delta \beta_1}, \frac{\delta f}{\delta \beta_2})$

$\Delta\beta = (\Delta\beta_1, \Delta\beta_2)$

(4) can be re written as :

$$\Delta f = \nabla f \cdot \Delta\beta \qquad (5)$$

This is not some expression that you haven't seen before , in fact we use it in our daily lives all the time. This expression simply means that , if we change our parameter by some amount , change in function can be calculated by multiplying change in parameter with gradient of the function w.r.t. the parameter.

Where do we use that in real life ?

When somebody asks you to find how much distance you covered if you were travelling at 4km/minute for 20 minutes .

you simply tell them 80km!, here distance is nothing but function of time ,

$\Delta f = 80km$ & $\Delta \beta = 20mins$, the speed is nothing but rate of change of distance w.r.t. time, or in other words , gradient of distance w.r.t. time $\nabla f = 4km/minute$ [gradient is nothing but rate of change !!]

You must be wondering , why are we talking about all of this ? eq (5) is magical . This gives us an idea about how we can change our parameters such that cost function always goes down

Consider

$$\Delta \beta = -\eta \nabla f \qquad (6)$$

Where η is some positive constant . if we put this back in (5), lets see what happens

$$\begin{aligned} \Delta f &= \nabla f \cdot \Delta \beta \\ &= -\eta \nabla f \cdot \nabla f \\ &= -\eta ||\nabla f||^2 \end{aligned} \qquad (7)$$

(7) is an amazing result , it tells us that **<u>if we changed our paramter , as per the suggestion in (6) , change in function will always be negative .</u>** This gives us a consistent way of changing our parameters so that our cost function always goes down. Once we start to reach near the optimal value, gradient of the cost function ∇f will tend to zero and our parameter will stop to change .

Now we have consistent method for starting from random values of βs and changing them in such a way that we eventually arrive at optimal values of βs for given historical data. Lets see whether it really works or not with an example in context of linear regression .

Linear Regression Parameter Estimation example with Gradient Descent

Recall our discussion on cost function for linear regression , it looked like this :

$$SSE = \sum {e_i}^2 = \sum (y_i - \hat{y_i})^2 = \sum (\, y_i - \beta_0 - \beta_1 * x_i)^2$$

If we consider entire X ,Y data and parameters βs to be written in matrix format like this

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ y_n \end{bmatrix}$$

$$X = \begin{bmatrix} 1 & x_{11} & x_{21} & . & . & x_{p1} \\ 1 & x_{12} & x_{22} & . & . & x_{p2} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ 1 & x_{1n} & x_{2n} & . & . & x_{pn} \end{bmatrix}$$

$$\beta = \begin{bmatrix} \beta_0 \\ \beta_1 \\ . \\ . \\ \beta_p \end{bmatrix}$$

We can write :

predictions = $X\beta$ [keep in mind that this will be a matrix multiplication]

errors = Y - predictions = $Y - X\beta$

cost $= (Y - X\beta)^T(Y - X\beta) = errors^T error$

I have taken liberty to extend the idea to more features which you see here as part of X matrix. x_{ij} represents j^{th} value of i^{th} variables/feature. All 1 column in X matrix represent multipliers of β_0.

Lets see what is the gradient of the loss function . Keep in mind that y_i and x_{ij} here are numbers/data points from the data. They are not parameters. They are observed values of the features.

gradient = $\nabla f = -2X^T(Y - X\beta)$

Lets simulate some data and put gradient descent to test

```
import pandas as pd
import numpy as np
```

```
x1=np.random.randint(low=1,high=20,size=20000)
x2=np.random.randint(low=1,high=20,size=20000)
```

```
y=3+2*x1-4*x2+np.random.random(20000)
```

you can see that we have generated data such that y is an approximate linear combination of x1 and x2, next we'll calculate optimal parameter values using gradient descent and compare them with results from sklearn and we'll see how good is the method.

```
x=pd.DataFrame({'intercept':np.ones(x1.shape[0]),'x1':x1,'x2':x2})
w=np.random.random(x.shape[1])
```

Lets write functions for predictions, error, cost and gradient that we discussed above

```
def myprediction(features,weights):
    predictions=np.dot(features,weights)
    return(predictions)

myprediction(x,w)
```

```
array([ 3.83044239,  5.51284694,  8.82009525, ...,  4.82993684,
       10.15713504,  2.49340259])
```

Note that , `np.dot` here is being used for matrix multiplication . Simple multiplication results to element wise multiplication , which is simply wrong in this context .

```
def myerror(target,features,weights):
    error=target-myprediction(features,weights)
    return(error)
myerror(y,x,w)
```

```
array([ -8.34075913, -26.3578085 , -53.60220599, ..., -39.73305801,
       -22.3456587 , -39.18996579])
```

```
def mycost(target,features,weights):
    error=myerror(target,features,weights)
    cost=np.dot(error.T,error)
    return(cost)

mycost(y,x,w)
```

```
23139076.992828812
```

```
def gradient(target,features,weights):

    error=myerror(target,features,weights)
    gradient=-np.dot(features.T,error)/features.shape[0]
    return(gradient)

gradient(y,x,w)
```

> array([24.86385998, 212.05914008, 369.58961913])

Note that gradient here is vector of 3 values because there are 3 parameters . Also since this is being evaluated on the entire data, we scaled it down with number of observations . Do recall that , the approximation which led to the ultimate results was that change in parameters is small. We don't have any direct control over gradient , we can always chose a small value for η??? to ensure that change in parameter remains small. Also if we end up choosing too small value for η???, we'll need to take larger number of steps to change in parameter in order to arrive at the optimal value of the parameters

Lets looks at the expected value for parameters from sklearn . Don't worry about the syntax here , we'll discuss that in detail, when we formally start with linear models in next module .

```
from sklearn.linear_model import LinearRegression
lr=LinearRegression()
lr.fit(x.iloc[:,1:],y)
sk_estimates=([lr.intercept_]+list(lr.coef_))
```

```
sk_estimates
```

> [3.4963871364502594, 2.0000361062367253, -3.999828135637543]

When you run the same , these might be different for you, as we generated the data randomly .Now lets write our version of this , using gradient descent

```
def my_lr(target,features,learning_rate,num_steps,print_when):

    # start with random values of parameters
    weights=np.random.random(features.shape[1])
    # change parameter multiple times in sequence
    # using the cost function gradient which we discussed earlier
    for i in range(num_steps):
        weights -= learning_rate*gradient(target,features,weights)
    # this simply prints the cost function value every (print_when)th iteration
        if i%print_when==0:
            print(mycost(target,features,weights),weights)

    return(weights)


```

```
my_lr(y,x,.0001,500,100)
```

```
result after 100th iteration : 29897491.344063997 [0.86957516 0.90950957
0.35280286]
result after 200th iteration : 5141497.711763546 [ 0.75555271  0.26200844
-1.71416641]
result after 300th iteration : 2703652.3603685475 [ 0.74787632  0.6560246
-2.37514143]
result after 400th iteration : 1480112.0586781118 [ 0.75044815  1.03096429
-2.77924166]
result after 500th iteration : 815173.9884761975 [ 0.75398277  1.31567619
-3.06931226]
```

```
final weights after 500 iterations: array([ 0.75755246,  1.52465372, -3.28073366])
```

you can see that if we take too few steps , we did not reach to the optimal value

Lets increase the learning rate η

```
my_lr(y,x,.01,500,100)
```

```
result after 100th iteration : 40621523.14640909 [ 0.2279513  -1.98371006
-3.60180851]
result after 200th iteration : 8.399918899418999e+30 [-8.28197980e+10
-9.56164546e+11 -9.47578310e+11]
result after 300th iteration : 2.0273152258521542e+54 [-4.06871492e+22
-4.69738039e+23 -4.65519851e+23]
result after 400th iteration : 4.892912746164989e+77 [-1.99885070e+34
-2.30769721e+35 -2.28697438e+35]
result after 500th iteration : 1.1809014620072595e+101 [-9.81981827e+45
-1.13370985e+47 -1.12352928e+47]
```

```
final weights after 500 iterations:array([3.68564341e+57, 4.25511972e+58,
4.21690928e+58])
```

You can see that because of high learning rate , change is parameter is huge and we end up missing the optimal point , cost function values , as well as parameter values ended up exploding. Now lets run with low learning rate and higher number of steps

```
my_lr(y,x,.0004,100000,10000)
```

```
result after 10000th iteration : 30786777.19050019 [0.17828571 0.48834937
0.70073856]
result after 20000th iteration : 12603.561197114073 [ 1.44895395  2.0897855
-3.91144197]
result after 30000th iteration : 5543.348390660241 [ 2.27839978  2.05342668
-3.94724853]
result after 40000th iteration : 3044.822379129174 [ 2.77182469  2.03179736
-3.96854931]
result after 50000th iteration : 2160.623502773709 [ 3.06535578  2.0189304
-3.98122083]
result after 60000th iteration : 1847.715952746199 [ 3.23997303  2.01127604
-3.98875893]
result after 70000th iteration : 1736.981662664487 [ 3.34385023  2.00672257
-3.99324323]
result after 80000th iteration : 1697.7941039148957 [ 3.40564521  2.00401379
-3.99591087]
result after 90000th iteration : 1683.926089232013 [ 3.44240612  2.00240237
-3.99749782]
result after 100000th iteration : 1679.018362458989 [ 3.46427464  2.00144376
-3.99844186]
```

```
final result after 100000th iteration :array([ 3.4772829 ,  2.00087354,
-3.99900342])
```

We can see here that we ended up getting pretty good estimates for β???s , as good as from sklearn .

```
sk_estimates
```

```
[3.4963871364502594, 2.0000361062367253, -3.999828135637543]
```

there are modifications to gradient descent in which can achieve the same thing in much less number of iterations. We'll discuss that in detail when we start with our course in Deep learning. For now ,we'll conclude this module here.

Chapter 4 : Linear Models

Our last module got little mathematical, and rightly so, for it forms the basis of much what follows next. However, That doesn't mean things are going to get even more complex. No, we'll focus more and more on practical aspects of ML as we move forward and see how the gap is bridged between the math and eventually its application in business.

You'll also notice that when it comes to application of things, its good to know math to understand whats going on at the back end; but its not absolutely necessary. You can always be "not so confident" about the math of things and still implement standard algorithms with much ease and accuracy with what we are going to learn next .

Meaning, you can always come back and have another go at making sense of all the mathematical jargon associated , but it shouldn't become a hurdle in you advancing through the implementation and usage of these algorithms. Persevere!

Lets summarize some relevant bits which we are going to refer to time and again.

There were two kinds of business problems :

1. Regression (with continuous values as target)
2. Classification (with [fixed]categorical values as target)

For Regression :

Predictive Model :

$$y_i \ = \ f(X_i)$$

Cost/Loss :

$$\sum_{i=1}^{i=n}(y_i \ - \ f(X_i))^2$$

For Classification :

Predictive Model :

$$p_i \ = \ \frac{1}{1+e^{-f(X_i)}}$$

Cost/Loss :

$$-\sum_{i=1}^{i=n}[y_i * log(p_i) + (1-y_i) * log(1-p_i)]$$

In both of these cases $f(X_i)$ can be any generic function, which we'll be referring to as Algorithms. In case of Linear Models , $f(X_i)$ is linear combination of input variables of the problem (for both regression and classification)

Formally; for linear models :

$$f(X_i) \ = \ \beta_0 \ + \ \beta_1 * x_{1i} \ + \ \beta_2 * x_{2i} \ + \ \beta_3 * x_{3i} \ldots\ldots$$

It seems our discussion on linear models should be finished here and we should move on to other parts of the course already. No, not really. There are many unanswered questions here before we start to build practical, industry level linear models. Here are those questions which we'll address one by one

1. All the discussions on theoretical aspects have made very convenient assumption that all the inputs for predicting the outcomes are going to be numeric, which is definitely not the case. For example , if we are trying to predict sales of different shops; part of a retail chain, which city they are located in can be a very important feature to consider. However in our discussion so far, we

have not considered how we convert this inherently categorical information into numbers and use it in our algorithms.
2. How do I know, how good are my predictions. Nobody in industry will accept my solution because I developed it, they need to know how it will tentatively perform before it makes into production
3. It seems that, if I pass data on 100 inputs/variables to this algorithm to predict the outcome of interest, it will give coefficient for all the inputs. Meaning, all the inputs are going to be part of the predictive model or in other words; all of them will have some impact on my predictions; irrespective of some of them being junk inputs. Our algorithm should contain some way of either completely removing them from the model or suppress their impact on our predictions.

First Question : Handling Categorical Variable

We'll start with the first question . Categorical variables are converted to dummies is the long and short of it. Lets figure out why?

In the context of numeric variables we can say that if the variable changed by some Δx amount then, our prediction will change by some constant multiplied by Δx amount. However this concept of change by Δx amount , just doesn't exist in context of categorical variables. Consider a case where we were trying to build a predictive model for how much time people take to run a 100 Meter dash, given their age and what kind of terrain they run on [hilly/flat]. You cant really say that terrain changed by `flat-hill`. All that you can say that people running on a hilly terrain , on an average will take some more time, assuming effect of age remains same across the population. We can depict this difference by using two separate predictive equations for both terrains

$$Runtime_{hills} = 10 + 0.5 * Age + 5$$
$$Runtime_{flats} = 10 + 0.5 * Age$$

Note : constant used here are indicative values

You can see; these two equations mean that if two people of same age run on two different terrains , person running on hills will take 5 seconds more. Does that mean if I have 100 categories in my categorical variable, I'll have to make 100 different models ? Not really , we can easily combine them like this :

$$Runtime_{hills} = 10 + 0.5 * Age + 5 * Terrain_{hill}$$

This one equation represents both the above seen separate equations. Variable $Terrain_{hill}$ takes value 1, when terrain is `hill` and value 0 when terrain is `flat`. This is called a dummy/flag variable. It is also known as one hot encoded representation of categorical data. Notice that we had two categories in our categorical variable `Terrain` , But we need only one dummy variable to represent it numerically . In general if our categorical variable has `n` categories , we need only `n-1` dummies. Theoretically it doesn't matter which category we ignore while creating dummies for rest.

One last thing that you need to keep in mind is that, since categories have an average impact; for their estimated average impact from the data to be consistent/reliable; they need to have backing of good number of observations. Essentially we should ignore categories which have too few obs in the data. Is there a magic number which we should consider as minimum required number of observations ? No, there isn't . A good sane choice will do, you can even experiment with different numbers.

Second Question : Validation

We'll start this discussion on this question , with another one , `why are we building this predictive model ?`

Simple answer is that we want to make use of that model to make prediction on future data, ideally we'd like to know the performance before data. But there is no way to get that future data. We'll have to make do with whatever training data we have been given.

It doesn't make sense to check performance of the model on the same data it was trained on, Since the model has already seen the outcome , it of course will perform on the same data as well as possible. That can not be taken as measure of its performance on unseen data. There are two ways , both of which have their pros/cons.

Breaking Training Data into Two Parts:

This is one of the simpler ways , but not without its flaws. You break your training data randomly in two parts , build the model on one and test its performance on another. Generally the training data is broken into 80:20, larger part to be used for training and smaller one to test performance. But again that isn't a magic ratio. Idea is to keep a small sample separate from the training data, but not too small (so that it contains similar patterns from the overall data) but not too big (so as to not have different patterns from the overall data).

Pros : Its quick and easy way to validate model

Cons : We don't have good idea about , what can be optimal ratio which is balance between not too big or too small . This random sample might be a niche part of the data having very different pattern from the rest of the data and in that case , measured performance will not be a good representative of real performance of the model

Cross-Validation :

Instead of breaking it into two parts , we break it into `K` parts. A good value of K is in the range 10-15. You'll have a better understanding of `why`, once we are done with this discussion .

Problem with simply breaking data into two parts was that we cant ensure that smaller validation set really resembles the overall data. Performance results on this will vary from the real performance. To counter that we break data into K parts. And build K models , every time leaving one of the parts as validation sample. This gives us out of sample performance measures for these K parts. Instead of using one of them as representative tentative performance. We take the average of these performance measures . Upon averaging , variations will be canceled out and we'll have more representative measure of performance. We can also look at variance of this performance to asses how stable it is across data.

Taking K as 10 means, at a time 10% of the data is not involved in the modeling process. Its fair to say 90% of the data will still have same pattern as the overall data. This number goes up as we break our data into more parts by taking higher value of K, but that has its down side of increase number of models we built (increasing the time taken)

Pros : Gives better measure of performance along with variance as stability measure .

Cons : Its time/resource taking, might as well be in-feasible for very large datasets.

Lets build a simple linear regression model

We'll resume our discussion on the third and final question after this section . In this section we are going to revisit data preparation with pandas and learn about sci-kit learns interface for building machine learning models

Problem Statement : We have been given data for people applying for a loan to a peer-to-peer lending firm. The data contains details of loan application and eventually how much interest rate was offered to people by the platform. Our solution needs to be able to predict the interest rate which will be offered to people , given their application detail as inputs. Lets look at the training data given to us

```
# import for processing data
import pandas as pd
import numpy as np
# imports for suppressing warnings
import warnings
warnings.filterwarnings('ignore')
```

```
# provide complete path for the file which contains your data
# r at the beginning is used to ensure that path is considerd as raw string and
# you dont get unicode error because of special characters combined with \ or /
file=r'/Users/lalitsachan/Dropbox/0.0 Data/loan_data_train.csv'
ld_train=pd.read_csv(file)
```

```
ld_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2200 entries, 0 to 2199
Data columns (total 15 columns):
ID                              2199 non-null float64
Amount.Requested                2199 non-null object
Amount.Funded.By.Investors      2199 non-null object
Interest.Rate                   2200 non-null object
Loan.Length                     2199 non-null object
Loan.Purpose                    2199 non-null object
Debt.To.Income.Ratio            2199 non-null object
State                           2199 non-null object
Home.Ownership                  2199 non-null object
Monthly.Income                  2197 non-null float64
FICO.Range                      2200 non-null object
Open.CREDIT.Lines               2196 non-null object
Revolving.CREDIT.Balance        2197 non-null object
Inquiries.in.the.Last.6.Months  2197 non-null float64
Employment.Length               2131 non-null object
dtypes: float64(3), object(12)
memory usage: 257.9+ KB
```

Lets comment on each of these variables one by one , as to what we are going to do before we start building our model

ID : It doesn't make sense to include unique identifiers of the observation (ID vars) as input. We'll drop this column

Amount.Requested , Open.CREDIT.Lines, Revolving.CREDIT.Balance : Ideally these should have been numeric columns, but if you look at the type assigned , it is object type. They must have come as object type because of some odd strings in the data at one or more places . We'll convert them to numeric type.

Amount.Funded.By.Investors : This information, although present in the data, will not come with loan application. If we want to build a model for predicting Interest.Rate using loan application characteristics , then we can not include this information in our model. We'll drop this column

Interest.Rate, Debt.To.Income.Ratio: These come as object type again because of the % symbol contained in it. We'll first remove the % sign and then convert it to numeric type

```
ld_train['Interest.Rate'].head()
```

```
0    18.49%
1    17.27%
2    14.33%
3    16.29%
4    12.23%
```

```
ld_train['Debt.To.Income.Ratio'].head()
```

```
0   27.56%
1   13.39%
2    3.50%
3   19.62%
4   23.79%
```

State , Home.Ownership, Loan.Length, Loan.Purpose : We'll create dummies , ignoring categories with too few occurrences

```
ld_train['State'].value_counts(dropna=False).head()
# only partial results are shown. to see the full results , remove .head()
```

```
CA   376
NY   231
FL   149
TX   146
PA    88
```

```
ld_train['Home.Ownership'].value_counts(dropna=False).head()
```

```
MORTGAGE   1018
RENT        999
OWN         177
OTHER         4
NONE          1
```

```
ld_train['Loan.Length'].value_counts(dropna=False)
```

```
36 months   1722
60 months    476
.              1
NaN            1
```

```
ld_train['Loan.Purpose'].value_counts(dropna=False).head()
# only partial results are shown. to see the full results , remove .head()
```

```
debt_consolidation   1147
credit_card           394
other                 174
home_improvement      135
major_purchase         84
```

Monthly.Income , Inquiries.in.the.Last.6.Months : Leave as is

FICO.Range: This comes as object type because the value written as numeric ranges in the data. As such, we can convert this to dummies, but we'll not be using information contained in order of the values if we convert them to dummies . We'll instead take average of the given range using string processing .

```
ld_train['FICO.Range'].value_counts().head()
```

```
670-674   151
675-679   144
680-684   141
695-699   138
665-669   129
```

Employment.Length: This takes type object; because it takes numeric values written in words. We can again chose to work with it like a categorical variable, but then we'll end up losing information on the order of values .

```
ld_train['Employment.Length'].value_counts(dropna=False)
```

```
10+ years  575
< 1 year   229
2 years    217
3 years    203
5 years    181
4 years    162
1 year     159
6 years    134
7 years    109
8 years    95
NaN        69
9 years    66
.          1
```

Lets begin with these operations that we have decided :

```
# removing ID and Amount.Funded.By.Investors
ld_train.drop(['ID','Amount.Funded.By.Investors'],axis=1,inplace=True)
```

```
# Removing % signs from two columns
for col in ['Interest.Rate','Debt.To.Income.Ratio']:
    ld_train[col]=ld_train[col].str.replace("%","")
```

```
# converting columns to numeric with pandas

for col in ['Amount.Requested', 'Interest.Rate','Debt.To.Income.Ratio',
            'Open.CREDIT.Lines','Revolving.CREDIT.Balance']:
    ld_train[col]=pd.to_numeric(ld_train[col],errors='coerce')
```

```
# Processing FICO.Range
k=ld_train['FICO.Range'].str.split("-",expand=True).astype(float)
ld_train['fico']=0.5*(k[0]+k[1])
del ld_train['FICO.Range']
```

```
# Processing Employment.Length

ld_train['Employment.Length']=ld_train['Employment.Length'].str.replace('years','')
ld_train['Employment.Length']=ld_train['Employment.Length'].str.replace('year','')
ld_train['Employment.Length']=np.where(ld_train['Employment.Length'].str[0]=='<',0,
                                       ld_train['Employment.Length'])
ld_train['Employment.Length']=np.where(ld_train['Employment.Length'].str[:2]=='10',10,
                                       ld_train['Employment.Length'])
ld_train['Employment.Length']=pd.to_numeric(ld_train['Employment.Length'],errors='coerce')

```

```
# Creating dummies with frequency cutoff

cat_col=['State' , 'Home.Ownership', 'Loan.Length', 'Loan.Purpose']
# this will be done for each of the columns in cat_col
```

```
for col in cat_col :
    # calculate frequency of categories in the columns
    k=ld_train[col].value_counts(dropna=False)
    # ignoring categories with too low frequencies and then selecting n-1 to
create dummies for
    cats=k.index[k>50][:-1]
    # creating dummies for remaining categories
    for cat in cats:
        # creating name of the dummy column coresponding to the category
        name=col+'_'+cat
        # adding the column to data
        ld_train[name]=(ld_train[col]==cat).astype(int)
    # removing the original column once we are done creating dummies for it
    del ld_train[col]
```

```
ld_train.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2200 entries, 0 to 2199
Data columns (total 31 columns):
Amount.Requested                    2195 non-null float64
Interest.Rate                       2200 non-null float64
Debt.To.Income.Ratio                2199 non-null float64
Monthly.Income                      2197 non-null float64
Open.CREDIT.Lines                   2193 non-null float64
Revolving.CREDIT.Balance            2195 non-null float64
Inquiries.in.the.Last.6.Months      2197 non-null float64
Employment.Length                   2130 non-null float64
fico                                2200 non-null float64
State_CA                            2200 non-null int64
State_NY                            2200 non-null int64
State_FL                            2200 non-null int64
State_TX                            2200 non-null int64
State_PA                            2200 non-null int64
State_IL                            2200 non-null int64
State_GA                            2200 non-null int64
State_NJ                            2200 non-null int64
State_VA                            2200 non-null int64
State_MA                            2200 non-null int64
State_NC                            2200 non-null int64
State_OH                            2200 non-null int64
State_MD                            2200 non-null int64
State_CO                            2200 non-null int64
Home.Ownership_MORTGAGE             2200 non-null int64
Home.Ownership_RENT                 2200 non-null int64
Loan.Length_36 months               2200 non-null int64
Loan.Purpose_debt_consolidation     2200 non-null int64
Loan.Purpose_credit_card            2200 non-null int64
Loan.Purpose_other                  2200 non-null int64
Loan.Purpose_home_improvement       2200 non-null int64
Loan.Purpose_major_purchase         2200 non-null int64
dtypes: float64(9), int64(22)
memory usage: 532.9 KB
```

All of the columns in the data are now numeric. Just one more thing to take care of before we start with modeling process. We need to make sure that there are no missing values in the data. if there are , then they need to be replaced .

```
# checking for missing values in the data
ld_train.isnull().sum()
```

```
Amount.Requested                    5
Interest.Rate                       0
Debt.To.Income.Ratio                1
Monthly.Income                      3
Open.CREDIT.Lines                   7
Revolving.CREDIT.Balance            5
Inquiries.in.the.Last.6.Months      3
Employment.Length                  70
fico                                0
State_CA                            0
State_NY                            0
State_FL                            0
State_TX                            0
State_PA                            0
State_IL                            0
State_GA                            0
State_NJ                            0
State_VA                            0
State_MA                            0
State_NC                            0
State_OH                            0
State_MD                            0
State_CO                            0
Home.Ownership_MORTGAGE             0
Home.Ownership_RENT                 0
Loan.Length_36 months               0
Loan.Purpose_debt_consolidation     0
Loan.Purpose_credit_card            0
Loan.Purpose_other                  0
Loan.Purpose_home_improvement       0
Loan.Purpose_major_purchase         0
dtype: int64
```

```
# imputing missing values with averages of the columns
for col in ld_train.columns:
    if ld_train[col].isnull().sum()>0:
        ld_train.loc[ld_train[col].isnull(),col]=ld_train[col].mean()

```

First we'll use the first method of validation , where we break data into two parts before start building the model

```
# breaking data into two parts
from sklearn.model_selection import train_test_split
t1,t2=train_test_split(ld_train,test_size=0.2,random_state=123)
# test_size=0.2, means the data is being split into two parts in the 80:20 ratio.
# t1 will contain 80%, and t2 will get 20% of the obs.
# random_state=123, simply makes the random process reproducible
```

we need to separate predictors (input/x vars) and target before we pass them to scikit-learn functions for building our linear regression model

```
x_train=t1.drop('Interest.Rate',axis=1)
y_train=t1['Interest.Rate']
x_test=t2.drop('Interest.Rate',axis=1)
y_test=t2['Interest.Rate']
```

Lets build our first linear regression model

```
# import the function for Linear Regression
from sklearn.linear_model import LinearRegression
lr=LinearRegression()
# fit function , builds the model ( parameter estimation etc)
lr.fit(x_train,y_train)
```

Lets looks at estimated coefficients . intercept is same as β_0???

```
lr.intercept_
```

```
76.32100980688705
```

```
list(zip(x_train.columns,lr.coef_))
```

```
[('Amount.Requested', 0.00016206823832747567),
('Debt.To.Income.Ratio', -0.005217635523226176),
('Monthly.Income', -4.0339147296499624e-05),
('Open.CREDIT.Lines', -0.03015894666977517),
('Revolving.CREDIT.Balance', -1.7860242337434248e-06),
('Inquiries.in.the.Last.6.Months', 0.32786067084992604),
('Employment.Length', 0.02325230583339998),
('fico', -0.08716732836779102),
('State_CA', -0.16231739562106098),
('State_NY', -0.14426278883807817),
('State_FL', -0.11716306311499997),
('State_TX', 0.4481165264861161),
('State_PA', -0.9332596674212796),
('State_IL', -0.4048740473139449),
('State_GA', -0.33202157322249337),
('State_NJ', -0.49634957660360035),
('State_VA', -0.13349751801583823),
('State_MA', -0.1634714204731154),
('State_NC', -0.47136779712009375),
('State_OH', -0.40429922213664504),
('State_MD', -0.1292878863756837),
('State_CO', 0.10071894446013128),
('Home.Ownership_MORTGAGE', -0.5636395222756556),
('Home.Ownership_RENT', -0.27130802518538744),
('Loan.Length_36 months', -3.1821676438146373),
('Loan.Purpose_debt_consolidation', -0.482384755055442),
('Loan.Purpose_credit_card', -0.5726731705822421),
('Loan.Purpose_other', 0.35159491851815755),
('Loan.Purpose_home_improvement', -0.4952547468027438),
('Loan.Purpose_major_purchase', -0.2391664596860732)]
```

Lets make predictions for performance check.

```
predicted_values=lr.predict(x_test)
from sklearn.metrics import mean_absolute_error
mean_absolute_error(predicted_values,y_test)
```

```
1.6531699740032333
```

This means that, we can assume that tentatively our model, will be off by 1.65 units on an average; while predicting interest rates on the basis of loan application. Now lets see, how we can do cross-validation with sklearn tools . Keep in mind that we don't need to break our data in this process. sklearn function will take care of that internally . All that we need to do is to separate target and predictors .

```
x_train=ld_train.drop('Interest.Rate',axis=1)
y_train=ld_train['Interest.Rate']
```

```
from sklearn.model_selection import cross_val_score
```

```
errors =
np.abs(cross_val_score(lr,x_train,y_train,cv=10,scoring='neg_mean_absolute_error'))
# cv=10 , means 10 fold cross validation
# Regarding scoring functions, the general theme in scikit learn is , higher the
better
# to remain consistent with the same , instead of mean_absolute_error, available
function for regression is neg_mean_absolute_error
# we can always wrap that and take positve values [ with np.abs ]
```

```
errors
```

```
[1.72994607, 1.70800508, 1.75246664, 1.63094103, 1.43860407,
 1.63204347, 1.42609575, 1.56465166, 1.53020947, 1.63953272]
```

```
avg_error=errors.mean()
error_std=np.std(errors)
avg_error,error_std
```

```
(1.6052495976597005, 0.10838823631170523)
```

Making Predictions with Real Test Data

What if we were given a separate test file where we eventually had to make prediction on and then submit (like projects). Biggest challenge will be to ensure that , test data eventually needs to have same columns as transformed training data on which we built the model.

Note : you can not check performance on this test data, if there is no response column given .

There are two ways to do so :

1. Combine training and test from the very beginning and then separate once the data prep is done. Build model on train and make predictions on test
2. Build a data-prep pipeline which gives same results for both train and test (We'll learn about this in later modules)

In here we'll give you a quick summary of the first method .

```
#### Combining two data sets
test_file=r'/Users/lalitsachan/Dropbox/0.0 Data/loan_data_test.csv'
ld_test=pd.read_csv(test_file)
```

```
# add an identifier column to both files so that they can be separated later on

ld_test['data']='test'
ld_train['data']='train'

# combine them
ld_all=pd.concat([ld_train,ld_test],axis=0)

# carry out the same data prep steps on ld_all as we did for ld_train

#~~~~~~~~~~ data prep on the ld_all~~~~~~~~~~~~~~

# make sure that you end up making dummies for column 'data'
# Now separate them

```

```
ld_train=ld_all[ld_all['data']=='train']
ld_test=ld_all[ld_all['data']=='test']

ld_test.drop(['data','Interest.Rate'],1,inplace=True)
del ld_train['data']
del ld_all
```

Now you can build model on `ld_train`, use the same model to make prediction on `ld_test` and make submission. Since both the datasets have gone through the same data prep process , they'll have same columns in them

Third Question : How to suppress effect of junk vars

Lets try to understand interpretation of βs .For a model which is written as :

$$f(X_i) = \beta_0 + \beta_1 * x_{1i} + \beta_2 * x_{2i} + \beta_3 * x_{3i} \ldots..$$

Lets pick x_2??? to comment on. The coefficient β_2???, associated with x_2???, simply means that if x_2??? changes by 1 unit, my prediction will change by β_2??? units. if β_2??? is +ve , then my prediction will increase as x_2??? increases; if β_2??? is -ve, my prediction will decrease when x_2??? increases [pointing to negative linear correlation].

But if x_2??? was a junk variable, it should not have any impact on my prediction. In order for that to happen, the coefficient associated with it should be zero or very close to zero.

what happens when we add a random variable to our model

Assuming you have a linear regression model, for easy notation consider first one, then two variables. This generalizes to two sets of models.
The first model is

$$I: y_i = \beta_0 + \beta_1 x_{1i} + \epsilon_i$$

the second model is

$$II: y_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} + \epsilon_i$$

This is solved by minimizing sum of squared residuals, for model one we want to minimize
$\text{SSR}_1 = \sum_i (y_i - \beta_0 - \beta_1 x_{1i})^2$???
and for model two we want to minimize

$\text{SSR}_2 = \sum_i (y_i - \beta_0 - \beta_1 x_{1i} - \beta_2 x_{2i})^2$???.

Lets say we have found the correct estimators for model 1, then you can obtain that exact same residual sum squares in model two by choosing the same values for β_0, β_1 and letting $\beta_2 = 0$. Now you can find, possibly, a lower sum squares residual by searching for a better value for β_2.

To summarize, the models are nested, in the sense that everything we can model with model 1 can be matched by model two, model two is more general than model 1. So, in the optimization, we have larger freedom with model two; so can always find a better solution.

This has really nothing to do with statistics but is a general fact about optimization. It extrapolates to following conclusion : If we add any variable in our model/data [junk or not junk]; cost function will always decrease . However for junk variables, it will go down by a small amount, and for good variables [which really do impact our target], it goes down by a large amount.

Regularisation

Problem with having junk vars a coefficient is that , their effect on the model doesn't remain consistent . Having lot of junk vars might lead to a model which performs entirely differently on training data and eventual validation/test data. That defies the purpose of building the model in the first place . This situation where our prediction model performs very well on the training data but

not so well on test/validation set; is called `Overfitting`. Ways to reduce this problem and make our model more generalizable is called `Regularisation` .

Many at times this is achieved by modifying our cost/loss formulation such that it reduces the impact of junk vars; and in general reduces impact of over-fitting by ensuring that our predictive model extracts most generic patterns from the data instead of simply memorizing training data.

We want to reduce impact of junk vars , that can be simply achieved by making their coefficient as close to zero as possible. We can achieve that by adding a penalty to the cost function on the size of βs . Lets understand how that works . Consider there are two variables x_{good} and x_{bad}. Both of them contribute to decrease in our traditional loss $L(\beta)$. x_{good} however results in higher decrease in comparison to x_{bad}

Lets say, contribution of x_{good} towards decrease of $L(\beta)$ is 10, where as for x_{bad}, it is 0.5 . Now we are going to add penalty to our loss formulation on the size of the parameters . Our new loss function will look like this :

$$L' = \text{Traditional Loss} + Penalty$$

this penalty can be anything as long as it serves the purpose to make our model more generalizable. We'll discuss some popular formulations for the same. But those by no means are the only ones which you are theoretically limited to use.

consider this one, known as L2 penalty [also known as `Ridge` regression in context of linear regression models] :

$$L' = L(\beta) + \lambda * \beta^2$$

We'll expand on the role of λ here in a bit , for now lets consider that to be 1, and lets also consider that; to start with coefficients for x_{good} and x_{bad}, both are 2 . Notice that penalty is always positive, meaning this will increase the loss function for all the vars, higher the parameter size [absolute value] , higher the increase in the loss. Now in the light of new loss formulation, decrease due to x_{good} will be = (10 - 2X2 = 6); and for x_{bad} it'll be = (0.5 - 2X2 = -3.5) .

Clearly this new loss formulation does not decrease because of x_{bad}, during optimization of loss, coefficient for x_{bad} will be moved to close to zero until decrease because of it becomes positive.

There are many such penalties we'll come across in our course discussions . They'll mainly be variations of the two, namely; L1 and L2 penalty.

cost with L1 penalty [also known as `Lasso`]: $L(\beta) + \lambda * |\beta|$???
cost with L2 penalty : $L(\beta) + \lambda * \beta^2$???

There are one basic impact of using either of the penalties to the cost function

L1 leads to model reduction but L2 doesnt

Lets consider case of having a single parameter β, same results get extrapolated to higher number of parameters also.

consider cost function with L2 penalty for linear regression (in matrix format as discussed in the earlier module) :

$$= (Y - X\beta)^T(Y - X\beta) + \lambda * \beta^2$$

We'll calculate the gradient and equate it to zero to determine our parameter value

$$\begin{aligned} &\nabla L = 0 \\ &=> -2 * X^T(Y - X\beta) + 2\lambda * \beta = 0 \\ &=> \beta = \frac{X^TY}{X^TX + \lambda} \end{aligned}$$

you can see that , by using higher and higher value of λ, you can make parameter to be very close to zero, but there is no way to make it absolutely zero.

Lets see what happens in case of using L1 penalty , here is the cost function with L1 penalty :

$$= (Y - X\beta)^T(Y - X\beta) + \lambda * |\beta|$$

Lets assume that β>0 for this discussion , you can do the exercise for -ve β as well and reach to the same conclusion. For +ve , cost function is :

$$= (Y - X\beta)^T(Y - X\beta) + \lambda * \beta$$

We'll equate the gradient to zero here also and lets see what happens

$$\nabla L = 0$$
$$=> -2 * X^T(Y - X\beta) + \lambda = 0$$
$$=> \beta = \frac{2X^TY - \lambda}{X^TX}$$

You can see that in this case , there does exist some value of λ??? for which parameter estimate can become 0

conclusion :

- Both L1 and L2 penalties reduce impact of junk vars on the model
- L1 penalty can result in some coefficient being exactly zero , thus model reduction
- Since both of them have penalty on the parameter size which is dependent on scale of variables , its advised that we should standardize the data if variables are on different scale

Chosing best value of λ with cross validaton

if we take λ=0, it simply leads to zero penalty and our traditional cost/loss formulation. If we take $\lambda \to \infty$, all βs will have to be zero, leading to complete reduction in the model (or no model). There is no mathematical formula for best value of λ, its different for different datasets. What we can do is , to try out different values of lambda and see its cross validated performance , and choose the one for which cross validated performance is best. Lets see how to do that [extending the example taken earlier] with sklearn functions

```
from sklearn.linear_model import Ridge,Lasso
from sklearn.model_selection import GridSearchCV
```

```
# we are going to try out values from 1 to 100
# what we have referring to as lambda is named alpha in the
# sklearn implementation
# we'll pass this dictionary to GridSearchCV function
lambdas=np.linspace(1,100,200)
params={'alpha':lambdas}
```

```
# this is the model for which we are tryin to estimate best value of lambda
model=Ridge(fit_intercept=True)
```

```
# this function will be trying out all the values of lambdas
# passed to it and record cross-validate performance
# using a custom function we'll extract the results
grid_search=GridSearchCV(model,param_grid=params,cv=10,scoring='neg_mean_absolute_error')
```

```
grid_search.fit(x_train,y_train)
```

```
grid_search.best_estimator_
```

> Ridge(alpha=13.43718592964824, copy_X=True, fit_intercept=True, max_iter=None, normalize=False, random_state=None, solver='auto', tol=0.001)

this is the best estimator [with best value of λ???], we can directly use this if we want or look at similar performance of other values and pick the one with least variance (most stable model) . Lets look at the custom function `Report` which will enable us to extract more detailed results from `grid_search.cv_results` object. Which is a huge dictionary containing information on performance of all combination of parameters that we are experiment with.

```
def report(results, n_top=3):
    for i in range(1, n_top + 1):
        # np.flatnonzero extracts index of `True` in a boolean array
        candidate = np.flatnonzero(results['rank_test_score'] == i)[0]
        # print rank of the model
        # values passed to function format here are put in the curly brackets when printing
        # 0 , 1 etc refer to placeholder for position of values passed to format function
        # .3f means upto 2 decimal digits
        print("Model with rank: {0}".format(i))
        # this prints cross validate performance and its standard deviation
        print("Mean validation score: {0:.5f} (std: {1:.5f})".format(
              results['mean_test_score'][candidate],
              results['std_test_score'][candidate]))
        # prints the paramter combination for which this performance was obtained
        print("Parameters: {0}".format(results['params'][candidate]))
        print("")
```

```
report(grid_search.cv_results_,3)
```

```
Model with rank: 1
Mean validation score: -1.60399 (std: 0.11152)
Parameters: {'alpha': 13.43718592964824}
```

```
Model with rank: 2
Mean validation score: -1.60399 (std: 0.11162)
Parameters: {'alpha': 13.93467336683417}
```

```
Model with rank: 3
Mean validation score: -1.60399 (std: 0.11171)
Parameters: {'alpha': 14.4321608040201}
```

Performance of these is pretty similar (looks identical due to us limiting this to 5 decimal digits) , there isn't much difference in stability either . We ca. go with the best guy. Tentative performance measure off by average 1.60 units .

if you want to make prediction , you can directly use `grid_search.predict`, it by defaults fits the model with best parameter choices on the entire data. However if you want to look at the coefficients, you'll have to fit the model separately.

```
ridge=grid_search.best_estimator_
```

```
ridge.fit(x_train,y_train)
```

```
ridge.intercept_
```

```
75.47046753682933
```

```
list(zip(x_train.columns,ridge.coef_))
```

```
[('Amount.Requested', 0.00016321376320470183),
('Debt.To.Income.Ratio', -0.0016649630043797535),
('Monthly.Income', -2.7907207451034204e-05),
('Open.CREDIT.Lines', -0.03648076179235473),
('Revolving.CREDIT.Balance', -2.683721318784198e-06),
('Inquiries.in.the.Last.6.Months', 0.3445032296978588),
('Employment.Length', 0.019601482704651712),
('fico', -0.08659781419319143),
('State_CA', -0.14115706864241717),
('State_NY', -0.10835699414412378),
('State_FL', -0.0112049655199493),
('State_TX', 0.4475943683966697),
('State_PA', -0.38750987879294396),
('State_IL', -0.47889963877418085),
('State_GA', -0.14818704990810666),
('State_NJ', -0.29109377437441625),
('State_VA', -0.05256231331588017),
('State_MA', -0.03973819224423696),
('State_NC', -0.342099870504766),
('State_OH', -0.202440726282369),
('State_MD', -0.023012728967286556),
('State_CO', 0.09019079073895374),
('Home.Ownership_MORTGAGE', -0.3571292206883735),
('Home.Ownership_RENT', -0.13033273805328766),
('Loan.Length_36 months', -3.013912446966353),
('Loan.Purpose_debt_consolidation', -0.41916847523850287),
('Loan.Purpose_credit_card', -0.5187875249317675),
('Loan.Purpose_other', 0.36593813217880045),
('Loan.Purpose_home_improvement', -0.3133150928985485),
('Loan.Purpose_major_purchase', -0.06049720536796199)]
```

you can see that there is no reduction in model coefficients . However if you compare them with coefficients obtained in simple linear regression without penalty, you'll find them that many of them have been suppressed by a good factor.

```
1 list(zip(x_train.columns,np.round(lr.coef_/ridge.coef_,2)))
```

```
[('Amount.Requested', 0.99),
('Debt.To.Income.Ratio', 3.13),
('Monthly.Income', 1.45),
('Open.CREDIT.Lines', 0.83),
('Revolving.CREDIT.Balance', 0.67),
('Inquiries.in.the.Last.6.Months', 0.95),
('Employment.Length', 1.19),
('fico', 1.01),
('State_CA', 1.15),
('State_NY', 1.33),
('State_FL', 10.46),
('State_TX', 1.0),
('State_PA', 2.41),
('State_IL', 0.85),
('State_GA', 2.24),
('State_NJ', 1.71),
('State_VA', 2.54),
('State_MA', 4.11),
('State_NC', 1.38),
```

```
('State_OH', 2.0),
('State_MD', 5.62),
('State_CO', 1.12),
('Home.Ownership_MORTGAGE', 1.58),
('Home.Ownership_RENT', 2.08),
('Loan.Length_36 months', 1.06),
('Loan.Purpose_debt_consolidation', 1.15),
('Loan.Purpose_credit_card', 1.1),
('Loan.Purpose_other', 0.96),
('Loan.Purpose_home_improvement', 1.58),
('Loan.Purpose_major_purchase', 3.95)]
```

Lets see if `Lasso`/`L1` penalty leads to model reduction or better performance .

we'll again do search for best λ for this

```
# there is no ideal range for any parameter search
# generally whatever range that you search in ,
# if the best value comes at the either edge of the range,
# you should expand on that side
# however if you keep on getting always largets value of lambda as best here
# that simply means that all vars are junk
# same on the lower side means , all vars are good and penalty is not necessary
lambdas=np.linspace(0.1,10,200)
params={'alpha':lambdas}
```

```
model=Lasso(fit_intercept=True)
```

```
grid_search=GridSearchCV(model,param_grid=params,cv=10,scoring='neg_mean_absolute_error')
```

```
grid_search.fit(x_train,y_train)
```

```
grid_search.best_estimator_
```

```
Lasso(alpha=0.1, copy_X=True, fit_intercept=True, max_iter=1000,
normalize=False, positive=False, precompute=False, random_state=None,
selection='cyclic', tol=0.0001, warm_start=False)
```

you can see that the best value came out to be on the lower end , we'll expand the range on that side .

```
lambdas=np.linspace(0.001,2,200)
params={'alpha':lambdas}
grid_search=GridSearchCV(model,param_grid=params,cv=10,scoring='neg_mean_absolute_error')
grid_search.fit(x_train,y_train)
```

```
grid_search.best_estimator_
```

```
Lasso(alpha=0.011045226130653268, copy_X=True, fit_intercept=True,
max_iter=1000, normalize=False, positive=False, precompute=False,
random_state=None, selection='cyclic', tol=0.0001, warm_start=False)
```

this value is in between the range, we're good . Lets look at the cross-validate performance of some of the top few models

```
report(grid_search.cv_results_,3)
```

> Model with rank: 1
> Mean validation score: -1.60128 (std: 0.11647)
> Parameters: {'alpha': 0.011045226130653268}

> Model with rank: 2
> Mean validation score: -1.60329 (std: 0.12013)
> Parameters: {'alpha': 0.021090452261306535}

> Model with rank: 3
> Mean validation score: -1.60411 (std: 0.10911)
> Parameters: {'alpha': 0.001}

There is no dramatic improvement in performance , now lets see if their is any model reduction (β =0)

```
lasso=grid_search.best_estimator_
lasso.fit(x_train,y_train)
```

```
list(zip(x_train.columns,lasso.coef_))
```

> [('Amount.Requested', 0.00016023851703962787),
> ('Debt.To.Income.Ratio', -0.0009942600081876468),
> ('Monthly.Income', -2.7304512141858614e-05),
> ('Open.CREDIT.Lines', -0.036990231977433084),
> ('Revolving.CREDIT.Balance', -2.8844495569785946e-06),
> ('Inquiries.in.the.Last.6.Months', 0.33452489786480466),
> ('Employment.Length', 0.015521998102599638),
> ('fico', -0.08654353305995562),
> ('State_CA', -0.0),
> ('State_NY', -0.0),
> ('State_FL', 0.0),
> ('State_TX', 0.4213181413677211),
> ('State_PA', -0.07076519997728142),
> ('State_IL', -0.16988773003610938),
> ('State_GA', -0.0),
> ('State_NJ', -0.0),
> ('State_VA', 0.0),
> ('State_MA', 0.0),
> ('State_NC', -0.0),
> ('State_OH', -0.0),
> ('State_MD', 0.0),
> ('State_CO', 0.0),
> ('Home.Ownership_MORTGAGE', -0.2085859940218407),
> ('Home.Ownership_RENT', -0.0),
> ('Loan.Length_36 months', -3.074433736924612),
> ('Loan.Purpose_debt_consolidation', -0.2540228534857545),
> ('Loan.Purpose_credit_card', -0.32972568683630343),
> ('Loan.Purpose_other', 0.3971422404793555),
> ('Loan.Purpose_home_improvement', -0.022332982889820444),
> ('Loan.Purpose_major_purchase', 0.0)]

you can see that many of the coefficients have become exactly zero . Much smaller model gives you similar performance . In this case, amongst all, Lasso model is the best , considering its size and performance. That doesn't mean, Lasso or L1 penalty will always result in best model. It depends on the data.

Note : Ridge and Lasso regression are just different ways of estimating the parameters. Eventual prediction model is linear in both as well as in simple linear regression

Lets build a classification linear model

You have seen how to model a continuous numeric response with linear regression technique. But in many business scenarios our target is binary. For example whether someone will buy my product , whether someone will default on the loan they have taken. Answer to all these and many other such questions is yes/no. Here the output variable values are discrete & finite rather than continuous & infinite values like in Linear Regression. For analyzing such type of data, we could try to frame a rule which helps in guessing the outcome from the input variables. This is called a classification problem which is an important topic in statistics and machine learning. classification is the problem of identifying to which of a set of categories a new observation belongs, on the basis of a training set of data containing observations whose category membership is known. Some examples of Classification Tasks are listed below:

- In medical field, the classification task could be assigning a diagnosis to a given patient as described by observed characteristics of the patient such as age, gender, blood pressure, body mass index, presence or absence of certain symptoms, etc.
- In banking sector, one may want to categorize hundreds or thousands of applications for new cards containing information for several attributes such as annual salary, outstanding debts, age etc., into users who have good credit or bad credit for enabling a credit card company to do further analysis for decision making; OR one might want to learn to predict whether a particular credit card charge is legitimate or fraudulent.
- In social sciences, we may be interested to predict the preference of a voter for a party based on : age, income, sex, race, residence state, votes in previous elections etc.
- In finance sector, one would require to ascertain , whether a vendor is credit worthy?
- In insurance domain, the company will need to assess , Is the submitted claim fraudulent or genuine?
- In Marketing, the marketer would like to figure out , Which segment of consumers are likely to buy?

All of the problems listed above use same underlying algorithms, despite them being pretty different from each other on the face of it.

In this module we'll look at where , $f(X)$ is linear combination of variables as discussed earlier . Here is the problem statement we'll be working on

Problem Statement :

A financial institution is planning to roll out a stock market trading facilitation service for their existing account holders. This service costs significant amount of money for the bank in terms of infra, licensing and people cost. To make the service offering profitable, they charge a percentage base commission on every trade transaction. However this is not a unique service offered by them, many of their other competitors are offering the same service and at lesser commission some times. To retain or attract people who trade heavily on stock market and in turn generate a good commission for institution, they are planning to offer discounts as they roll out the service to entire customer base.

Problem is , that this discount, hampers profits coming from the customers who do not trade in large quantities . To tackle this issue , company wants to offer discounts selectively. To be able to do so, they need to know which of their customers are going to be heavy traders or money makers for them.

To be able to do this, they decided to do a beta run of their service to a small chunk of their customer base [approx 10000 people]. For these customers they have manually divided them into two revenue categories 1 and 2. Revenue one category is the one which are money makers for the bank, revenue category 2 are the ones which need to be kept out of discount offers.

We need to use this study's data to build a prediction model which should be able to identify if a customer is potentially eligible for discounts [falls In revenue grid category 1]. Lets get the data and begin.

Classification Model Evaluation and Probability Scores to Hard Classes

Before we jump-in head first into building our model, we need to figure out couple of things about classification problems in general . We discussed earlier that prediction model for classification output probabilities as outcome. In many case that'll suffice as a way of scoring our observation in terms most likely to least likely. However in many cases we'd eventually need to convert those probability scores to hard classes.

The way is to figure out a cut-off/threshold for the probability scores where we can say that, obs with higher score than this will be classified as 1 and others will be classified as 0. Now, the question becomes; how do we come up with this cut-off.

Confusion Matrix and associated measurements

Irrespective of what cut-off we chose, the hard class prediction rule is fixed. if probability score is higher than the cutoff then the prediction is 1 otherwise 0 [given, we are predicting probability of outcome being 1]. Any cut-off decision results in some of our predictions being true and some of them being false. Since there are are only two hard classes, we'll have 4 possible cases .

- When we predict 1 [+ve] but in reality the outcome is 0[-ve] : False Positive
- When we predict 0 [-ve] but in reality the outcome is 1[+ve] : False Negative
- When we predict 1 [+ve] and in reality the outcome is 1[+ve] : True Positive
- When we predict 0 [-ve] and in reality the outcome is 0[-ve] : True Positive

This is generally displayed in a matrix format known as confusion matrix .

**	**Positive_predicted**	**Negative_predicted**
Positive_real	**TP**	**FN**
Negative_real	**FP**	**TN**

TP : Number of true positive cases
TN : Number of true negative cases
FP : Number of false positive cases
FN : Number of false negative cases

Using these figures we can come up with couple of popular measurements in context of classification

How accurate our predictions are

Accuracy = $\frac{TP+TN}{\text{Total obs}}$???

How well we are capturing/recalling positive cases

Sensitivity or Recall = $\frac{TP}{TP+FN}$

How well we are capturing negative cases

Specificity = $\frac{TN}{TN+FP}$

How accurately we have been able to predict positive cases

Precision = $\frac{TP}{TP+FP}$

Some might suggest that we can take any one of them, and measure for all scores , and decides that score to be our cutoff where the taken measurement is highest for the training data. However none of these above mentioned measurements take care of our business requirement of good separation between two classes. Here are the issues associated with these individually if we consider them as candidate for determining cutoffs .

Accuracy This works only if our classes are roughly equally present in the data. However generally this is not the case in many business problems. For example, consider campaign response model. Typical response rate is 0.5-2%. Even if predict that none of the customers are going to subscribe to our campaign; accuracy will be in the range 98-99.5%, quite misleading.

Sensitivity or Recall We can arbitrarily make Sensitivity/Recall 100% by predicting all the cases as positive . Which is kinda useless because it doesn't take into account that labeling lot of negative cases as positive should be penalized too.

Specificity We can arbitrarily make Specificity 100% by predicting all the cases as Negative . Which is kinda useless because it doesn't take into account that labeling lot of positive cases as negative should be penalized too.

Precision We can make precision very high by keep cut-off too high and thus predicting very few sure shot cases as positive, but in that scenario our recall will be down to dumps.

So it turns out that, these measures are a good look into how our model is doing , but none of them taken individually can be used to determine proper cut-off. Following are some proper measures which give equal weight to goal of capturing as many positives as possible and at the same time; not labeling too many negative cases as positives .

KS = $\frac{TP}{TP+FN} - \frac{FP}{TN+FP}$???

You can see that `KS` will be highest when we recall is high but at the same time we are not labeling a lot of negative cases as positive. We can calculate `KS` for all the scores and chose that score as our ideal cut-off for which `KS` is maximum.

There is another measure which lets you give different weights for your precision or recall. There can be cases in real business problems where you'd want to give more importance to recall over precision or otherwise. Consider a critical test which determines whether someone has a particular aggressively fatal decease or not . In this case we wouldn't want to miss out on positive cases and wouldn't really mind some of the negative cases being labeled as positive.

F_β Score = $\frac{(1+\beta^2)*Precision*Recall}{(\beta^2*Precision)+Recall}$

Notice that value of β??? here will determine; how much and what are we going to give importance to. For β???=1 , equal importance is given to both precision and recall. When 1> $\beta \rightarrow 0$???, F_β??? score favors Precision and results in high probability score being chosen as cutoff. On the other hand when β??? > 1 and as $\beta \rightarrow \infty$??? it favors Recall and results in low probability scores being chooses as cutoff.

Now these let us find a proper cutoff given a probability score model. However so far we haven't discussed how do asses , how good the score is itself . Next section is dedicated to the same .

ROC curve and AUC Score

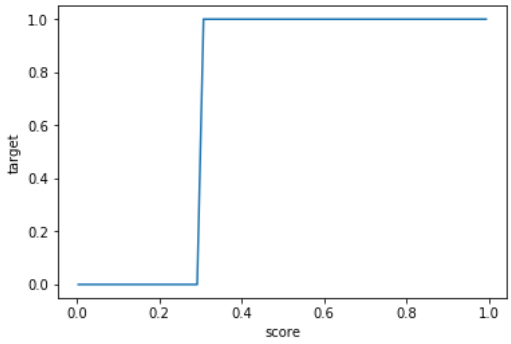
When we asses performance of a classification probability score , we try to see how it stacks up with an ideal scenario. For an ideal score, there will exist a clean cutoff; meaning, there will be no overlap between two classes when chose that cut-off. There will be no `False Positives` or `False Negatives`

Consider a hypothetical scenario where we have an ideal prob score like given below .

```
d=pd.DataFrame({'score':np.random.random(size=100)})
d['target']=(d['score']>0.3).astype(int)
```

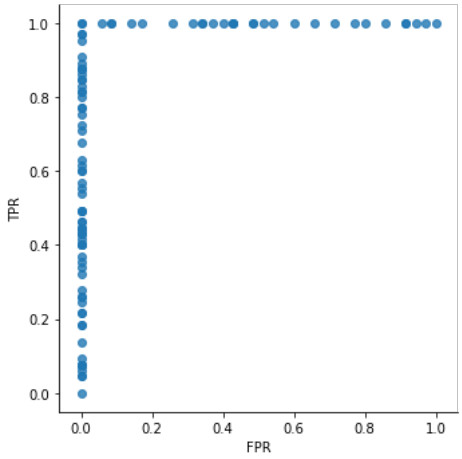
```
import seaborn as sns
```

```
sns.lineplot(x='score',y='target',data=d)
```



For this ideal scenario, we are going to consider many cutoffs between 0-1 and calculate `True Positive Rate` [Same as Sensitivity] and `False Positive Rate` [Same as 1-Specificity]. And plot those . The resultant plot that we'll get is known as ROC Curve . Lets see how that looks

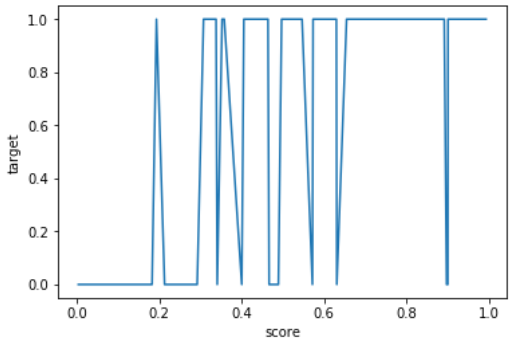
```
TPR=[]
FPR=[]
real=d['target']
for cutoff in np.linspace(0,1,100):
    predicted=(d['score']>cutoff).astype(int)
    TP=((real==1)&(predicted==1)).sum()
    FP=((real==0)&(predicted==1)).sum()
    TN=((real==0)&(predicted==0)).sum()
    FN=((real==1)&(predicted==0)).sum()

    TPR.append(TP/(TP+FN))
    FPR.append(FP/(TN+FP))

temp=pd.DataFrame({'TPR':TPR,'FPR':FPR})

sns.lmplot(y='TPR',x='FPR',data=temp,fit_reg=False)
```



Its perfectly triangular , and area under the curve is 1 for this ideal scenario, however lets see what happens if we make it not so ideal scenario and add some overlap. [we'll flip some targets in the same data]

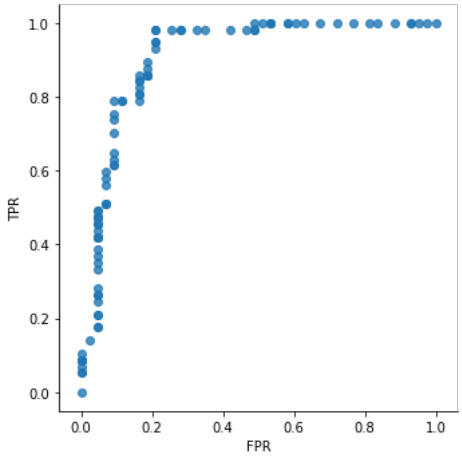
```
inds=np.random.choice(range(100),10,replace=False)
d.iloc[inds,1]=1-d.iloc[inds,1]
```

```
sns.lineplot(x='score',y='target',data=d)
```



you can see that there is lot of overlap now and , there can not exist a clear cutoff. Lets see how the ROC curve looks for the same

```
TPR=[]
FPR=[]
real=d['target']
for cutoff in np.linspace(0,1,100):
    predicted=(d['score']>cutoff).astype(int)
    TP=((real==1)&(predicted==1)).sum()
    FP=((real==0)&(predicted==1)).sum()
    TN=((real==0)&(predicted==0)).sum()
    FN=((real==1)&(predicted==0)).sum()

    TPR.append(TP/(TP+FN))
    FPR.append(FP/(TN+FP))

temp=pd.DataFrame({'TPR':TPR,'FPR':FPR})

sns.lmplot(y='TPR',x='FPR',data=temp,fit_reg=False)
```



You can try introducing more overlap and you'll see the ROC curve move away from the ideal scenario. Area Under the Curve(AUC Score) will become less than one, more close to 1 it is more close to ideal scenario it is, better is your model . Now that we have a way to asses our model , lets begin building it

```
# Here we we'll demonstrate our earlier suggestion, where we combine train
# test data and do data prep so that both of them have same set of variables
```

```
train_file=r'~/Dropbox/0.0 Data/rg_train.csv'
test_file=r'~/Dropbox/0.0 Data/rg_test.csv'
bd_train=pd.read_csv(train_file)

bd_test=pd.read_csv(test_file)
bd_train['data']='train'
bd_test['data']='test'
bd_all=pd.concat([bd_train,bd_test],axis=0)
```

These are few data decision that we have taken after exploring the data Feel free to go alternate routes and see how that makes a difference to model performance

```
# REF_NO,post_code , post_area  : drop [ too many unique values]
# children : Zero : 0 , 4+ : 4 and then convert to numeric
# age_band, family income : string processing and then to numeric
# status , occupation , occupation_partner , home_status,
# self_employed ,TVArea , Region , gender : dummies
# Revenue Grid : 1,2 : 1,0 [some functions need the target to be 1/0 in binary classfication]
```

```
bd_all.drop(['REF_NO','post_code','post_area'],axis=1,inplace=True)
```

```
bd_all['children']=np.where(bd_all['children']=='Zero',0,bd_all['children'])
bd_all['children']=np.where(bd_all['children'].str[:1]=='4',4,bd_all['children'])
bd_all['children']=pd.to_numeric(bd_all['children'],errors='coerce')
```

```
bd_all['Revenue.Grid']=(bd_all['Revenue.Grid']==1).astype(int)
```

```
bd_all['family_income'].value_counts(dropna=False)
```

```
'>=35,000              2517
<27,500, >=25,000      1227
<30,000, >=27,500       994
<25,000, >=22,500       833
<20,000, >=17,500       683
<12,500, >=10,000       677
<17,500, >=15,000       634
<15,000, >=12,500       629
<22,500, >=20,000       590
<10,000, >= 8,000       563
< 8,000, >= 4,000       402
< 4,000                 278
Unknown                 128
```

```
bd_all['family_income']=bd_all['family_income'].str.replace(',',"")
bd_all['family_income']=bd_all['family_income'].str.replace('<',"")
k=bd_all['family_income'].str.split('>=',expand=True)
```

```
for col in k.columns:
    k[col]=pd.to_numeric(k[col],errors='coerce')
```

```
bd_all['fi']=np.where(bd_all['family_income']=='Unknown',np.nan,
                      np.where(k[0].isnull(),k[1],
                      np.where(k[1].isnull(),k[0],0.5*(k[0]+k[1]))))
```

```
bd_all['age_band'].value_counts(dropna=False)
```

```
45-50    1359
36-40    1134
41-45    1112
31-35    1061
51-55    1052
55-60    1047
26-30     927
61-65     881
65-70     598
22-25     456
71+       410
18-21      63
Unknown    55
```

```
k=bd_all['age_band'].str.split('-',expand=True)
for col in k.columns:
    k[col]=pd.to_numeric(k[col],errors='coerce')
```

```
bd_all['ab']=np.where(bd_all['age_band'].str[:2]=='71',71,
                      np.where(bd_all['age_band']=='Unknow',np.nan,0.5*(k[0]+k[1])))
```

```
del bd_all['age_band']
del bd_all['family_income']
```

```
cat_vars=bd_all.select_dtypes(['object']).columns
cat_vars=list(cat_vars)
cat_vars.remove('data')
# we are using pd.get_dummies here to create dummies
# its more straight forward but doesnt let you ignore categories on the basis of frequencies
for col in cat_vars:
    dummy=pd.get_dummies(bd_all[col],drop_first=True,prefix=col)
    bd_all=pd.concat([bd_all,dummy],axis=1)
    del bd_all[col]
    print(col)
del dummy>
```

```
TVarea
gender
home_status
occupation
occupation_partner
region
self_employed
self_employed_partner
status
```

```
# imputing missing values
for col in bd_all.columns:
   if col=='data' or bd_all[col].isnull().sum()==0:continue

bd_all.loc[bd_all[col].isnull(),col]=bd_all.loc[bd_all['data']=='train',col].mean()

```

```
# separating data
bd_train=bd_all[bd_all['data']=='train']
del bd_train['data']
bd_test=bd_all[bd_all['data']=='test']
bd_test.drop(['Revenue.Grid','data'],axis=1,inplace=True)

```

In scikit-learn L1 and L2 penalties are implemented within the function `LogisticRegression` it self. We'll be treating them as parameters to tune with cross validation. The counterpart to λ is C, which is implemented in a way that , lower the value of C, higher the penalty . There is another parameter `class_weight`, takes two values `balanced` and `None`. `None` here implies equal weight to all observation while calculating the cost/loss [thus all observation having equal contribution to loss, this might make the model biased towards the majority class if their is class imbalance] . `balanced` artificially inflates weight of the minority class [class with low frequency] in order to ensure that cost/loss contribution is same from both the classes and model is focused on separation of the classes .

```
from sklearn.linear_model import LogisticRegression
```

```
params={'class_weight':['balanced',None],
        'penalty':['l1','l2'],
        # these are L1 and L2 written in lower case
        # dont confuse them with numeric eleven and tweleve
        'C':np.linspace(0.0001,1000,10)}
# we can certainly try much higher ranges and number of values for the parameter
'C'
# grid search in this case , will be trying out 2*2*10=40 possible combination
# and will give us cross validated performance for all
```

```
model=LogisticRegression(fit_intercept=True)
```

```
from sklearn.model_selection import GridSearchCV
```

```
grid_search=GridSearchCV(model,param_grid=params,cv=10,scoring="roc_auc",n_jobs=-1)
# note that scoring is now roc_auc as we are solving a classification problem
# n_jobs has nothing to do with model building as such
# it enables parallel processing , number reflects number of cores
# of your processor being utilised. -1 , means all the cores
```

```
x_train=bd_train.drop('Revenue.Grid',axis=1)
y_train=bd_train['Revenue.Grid']
```

```
grid_search.fit(x_train,y_train)
```

```
report(grid_search.cv_results_,3)
```

> Model with rank: 1
> Mean validation score: 0.95466 (std: 0.01253)
> Parameters: {'C': 0.0001, 'class_weight': 'balanced', 'penalty': 'l2'}

> Model with rank: 2
> Mean validation score: 0.95269 (std: 0.01273)
> Parameters: {'C': 555.5556, 'class_weight': 'balanced', 'penalty': 'l2'}

> Model with rank: 3
> Mean validation score: 0.95255 (std: 0.01325)
> Parameters: {'C': 444.4445, 'class_weight': 'balanced', 'penalty': 'l2'}

We'll go ahead with the best model here . Although ideally we should expand the range on C on the lower side and run the experiment as the best values is coming at the edge. I am leaving that for you to try . Since this is with `l2` penalty there will not be any model reduction

if we want to make prediction just for probabilities and submit , we can simply use `grid_search` object. As for the tentative performance of this model, we can already see that in the outcome of report function with cross validated auc score

```
# predict_proba for predciting probabilities
# just predict, predicts hard classes considering 0.5 as score cutoff
# which is not always a great idea, we'll see in a moment
# how to determine our own cutoff, in case we need to predict hard classes
test_prediction = grid_search.predict_proba(bd_test)
```

```
test_prediction
```

```
array([[0.99496285, 0.00503715],
       [0.95418665, 0.04581335],
       [0.98695233, 0.01304767],
       ...,
       [0.97058409, 0.02941591],
       [0.77185663, 0.22814337],
       [0.80655962, 0.19344038]])
```

Note that this gives two probabilities for each observation and they sum up to 1

```
# this will tell you which probability belongs to which class
grid_search.classes_
```

```
array([0, 1])
```

this means first probability is for the outcome being 0 and second is for the outcome being 1

you can extract probability for either class by using proper index

- `test_prediction[:,0]` : for probability of outcome being 0
- `test_prediction[:,1]` : for probability of outcome being 1

you can submit this by , converting it to a pandas data frame and then using function `to_csv` to write it to a csv file

Finding cutoff on the basis of max KS

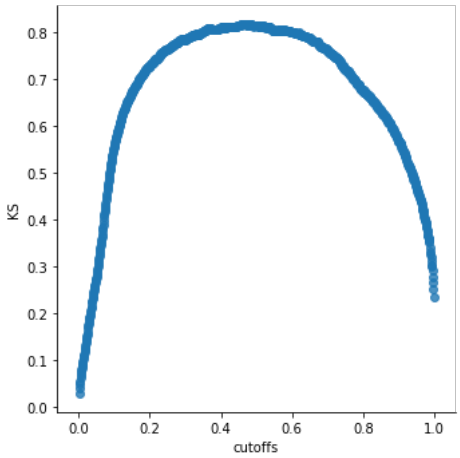
```
train_score=grid_search.predict_proba(x_train)[:,1]
real = y_train
```

```
cutoffs = np.linspace(.001,0.999, 999)
```

```
KS=[]
```

```
for cutoff in cutoffs:
    predicted=(train_score>cutoff).astype(int)
    TP=((real==1)&(predicted==1)).sum()
    FP=((real==0)&(predicted==1)).sum()
    TN=((real==0)&(predicted==0)).sum()
    FN=((real==1)&(predicted==0)).sum()

    ks=(TP/(TP+FN))-(FP/(TN+FP))
    KS.append(ks)
```

```
temp=pd.DataFrame({'cutoffs':cutoffs,'KS':KS})
sns.lmplot(x='cutoffs',y='KS',data=temp,fit_reg=False)
```



We can now find for which cutoff value KS takes its maximum value

```
cutoffs[KS==max(KS)][0]
```

> 0.467

if we have to submit hard-class we'll use this cutoff to convert probability score to hard classes

```
test_hard_classes=(test_prediction>cutoffs[KS==max(KS)][0]).astype(int)
```

and write this to csv to submit

Before we conclude this module, few key takeaways :

- Data Prep part will remains same , irrespective of what modules we study further
- Performance measures and methods will also remain same irrespective of the algorithm
- The way to find cutoffs for probability score will also remain same irrespective what which algorithm we obtain those scores from

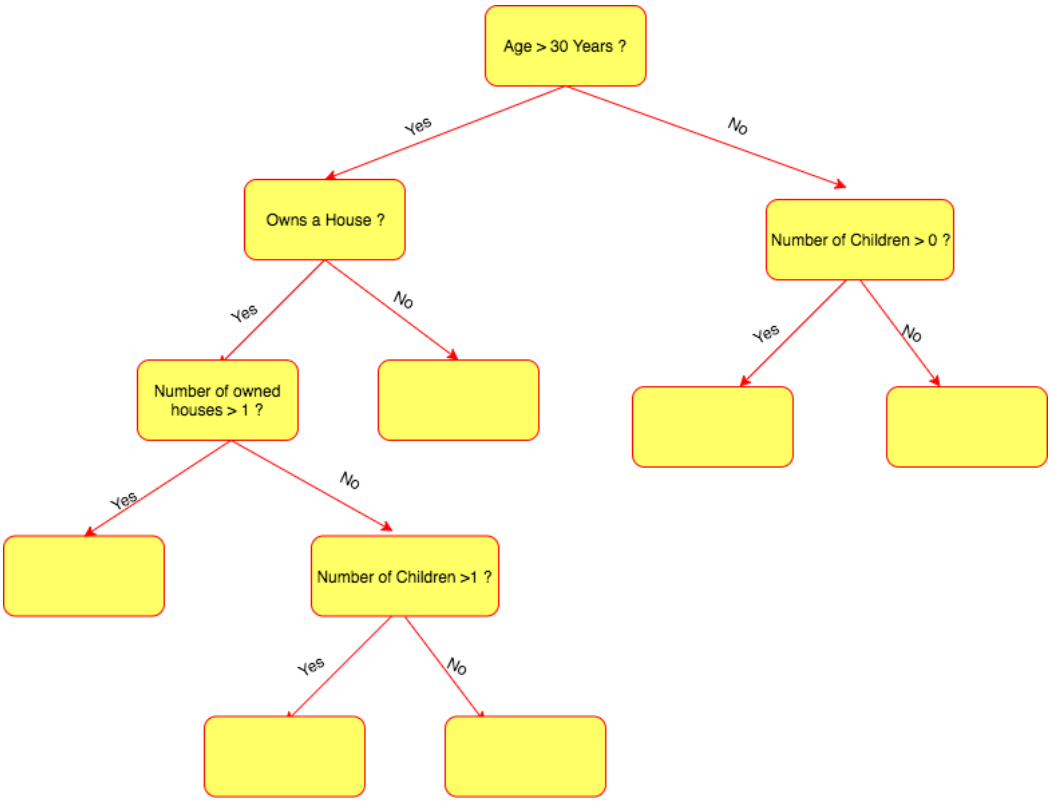
Chapter 5 : Decision Trees and Random Forests

We will start our discussion with Decision Trees. Decision trees are a hierarchical way of partitioning the data starting with the entire data and recursively partitioning it into smaller parts.

We will cover the following points in our discussion:

- Basic structure of a decision tree
- Building a classification tree (Regression tree is similar with some changes)
- Interpretation in the absence of coefficients
- Decision tree implementation
- Over-fitting with decision trees
- Random Forests
- Random Forest implementation

Lets start with a classification example of predicting whether someone will buy an insurance or not. We have been given a set of rules which can be shown as the diagram below:



What we see here is an example of a decision tree. A decision tree is drawn upside down with the root at the top.

Starting from the top, the first question asked is whether the a persons age is greater than 30 years or not. Depending on the answer, a second question is asked, either a person owns a house or not or does the person have 1 or more children and so on. Lets say that the person we consider is 45 years old, then the answer to the first question is Yes and then the next question for this person would be whether this person owns a house or not. Lets say he/she does not own a house. Now we end up in a node where no further questions are asked. This is the **terminal node**. In the terminal node, no further questions are asked. The nodes where we ask questions are the **decision nodes** with the top one being considered as the **parent node**. A thing to note here is that all the questions have binary answers - yes or no.

Now, we know that this person who is 45 years old and does not own a house ends up in one of the terminal nodes - we can say that this person belongs to this bucket. Now, our main question is to predict whether the person with these characteristics will buy the insurance or not.

Before we can answer this question, we need to understand a few more things:

1. **How do we make predictions if some observation ends up in the terminal node?**
2. **Where do these rules come from?**
3. **How do we pick rules to split each node?**
4. **When do we stop splitting a node and consider it as a terminal node?**

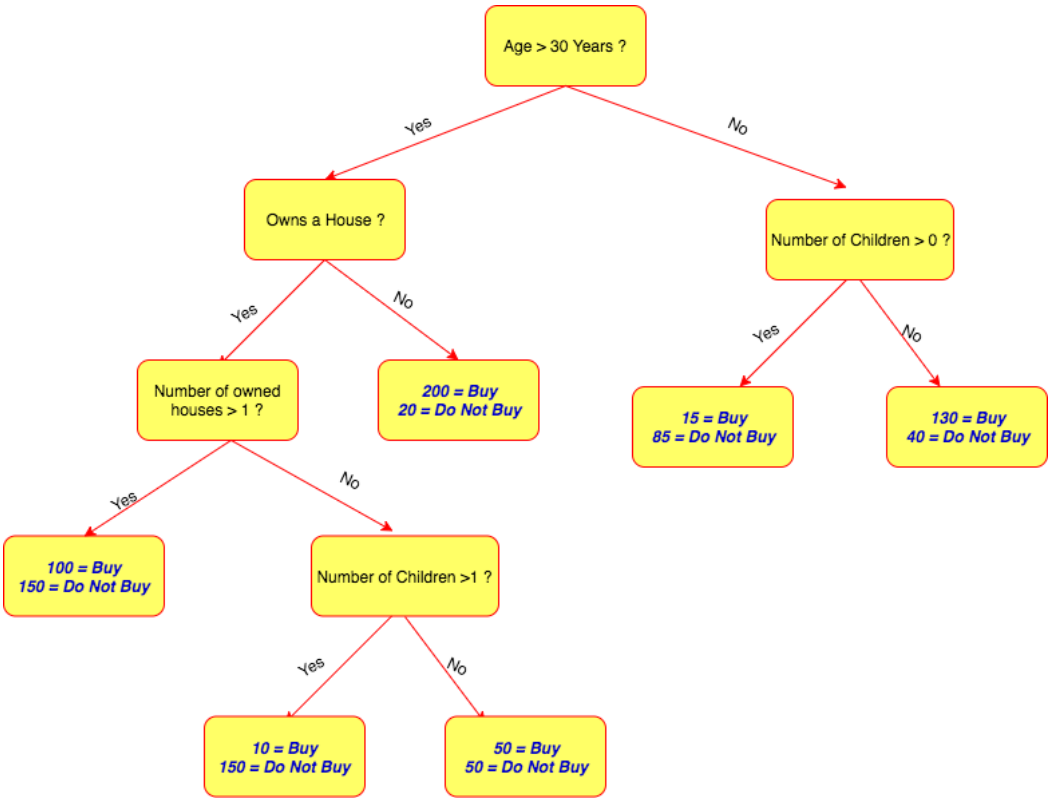
Once we come up with the answers for the questions above, we will have a better idea about decision trees.

1. How do we make predictions if an observation ends up in a terminal node?

The way predictions are made for a classification problem are different than the way they are made for a regression problem. The details are described below.

Classification:

Someone had given us the rules using which we made the decision tree shown above. Instead, we ask this person to give us the information using which he built the tree i.e. share the data using which he/she could come up with the rules. This data can also be referred to as the training data. We took this training data and passed each observation from this data through the decision tree, resulting in the following tree:



Note the terminal nodes now. We can see that among people with age greater than 30 years and who do not own a house, 200 of them bought the insurance and 20 did not buy. Among people with age less than 30 years and who have children, 15 people bought the insurance and 85 did not buy. Using this information present in the the terminal nodes, we can make prediction for new people for whom we do not know the outcome. For example, lets consider a new person comes whose age is greater than 30 years and does not own a house. Using our training data we saw that most of the people who end up in that node buy the insurance. So our prediction will be that this new person will buy the insurance using a simple majority vote. Instead of using the majority vote, we can also consider the probability of this person buying the insurance i.e. 200/220 = 0.9, which is quite high. We can say that if someone ends up in this terminal node, there is a high chance or probability of

this person to buy the insurance.

Now lets consider a person who owns 1 house and has no children, then this person ends in a node where 50% of the people buy the insurance and 50% don't. This probability of a person buying the insurance is as random as a coin flip and hence not desirable.

In short, predictions can be made using either of the following:

1. Probability:

   Given by proportions in the terminal nodes

2. Hard classes:

   - Simple majority vote
   - Cutoff on the probability score

Regression:

When the response variable is continuous, in order to make predictions, we take an average of the response values present in the terminal nodes.

Whether we have a classification decision tree or a regression decision tree, we now know how to make predictions.

But we still don't know how the decision tree was built in the first place. In order to figure this out, we need to answer the next question:

2. Where do these rules come from?

The rules are primarily binary questions. How do we come up with binary questions from numerical and categorical variables?

Numeric variables: For continuous variables, we simply discretize the range and ask questions on the intervals. Lets say we have values 5, 10, 15, 16 and 20 in a variable say 'age' as follows:

Questions like 'is age greater than 10' have an answer either 'yes' or 'no' which covers the entire data. Similarly, the rule 'is age greater than 15' will cover the entire range as well. Binary questions like these can cover the entire data range. Also, it is not necessary that the intervals in which we break the range should be equidistant. For example, in the line above, there is no value present between 10 and 15. The question 'is age greater than 11' and the question 'is age greater than 14' will result in the same partitioning of the data. This is because there are no observations for the variable 'age' in the range 10 to 15. So whatever question we ask between the range 10 to 15 will result in the same partition of the data.

The way a continuous variable is discretized and how the interval questions are decided depends on the kinds of values the continuous variable has.

Categorical variables:

We are already aware how to make dummy variables from categorical predictors. Lets consider the following dummy variables created for the 'City' predictor:

City	var_delhi	var_new_york	var_beijing
delhi	1	0	0
new york	0	1	0
beijing	0	0	1
new york	0	1	0
...	...	...	...
...	...	...	...
delhi	1	0	0

Using the categorical column 'City', we create three dummy variables. In decision trees, an example of a rule is 'Is var_delhi greater than 0.5' which is the same as asking whether it is 0 or 1 i.e if the variables value is greater than 0.5 then the value is delhi else it is either of the other cities.

This is how we get rules from numerical and categorical variables.

3. How to pick rules for splitting at each node?

We have understood how the rules are made for categorical and numerical variables. Given some features, the number of rules can be quite big. How do we choose the best rule amongst these to split a node. The way to pick rules would differ for classification and regression.

Classification:

Lets, for a moment, consider what an ideal decision tree would be like. Which decisions would we be most happy with? The decisions which gives a clear majority or in more specific words, a decision which results in a more homogeneous child node. Lets consider the example of a person who is 45 years old and does not own a home - whether this person will buy the insurance or not. 200 out of the 220 people present in the terminal node end up buying the insurance. Hence, if we get another person with similar characteristics, we can reasonably predict that that person would buy the insurance. Similarly, if a person ends up in a terminal node in which 10 out of 170 people usually end up buying the insurance, then we can be reasonably certain that this person will not buy the insurance. Both these cases are preferred since they result in a more homogeneous terminal node.

However, we would not be certain whether a person will end up buying the insurance if 50 out of 100 people ending up in the terminal node buy the insurance i.e. 50% of the people can either buy or not buy the insurance. The decision is as good as a coin flip.

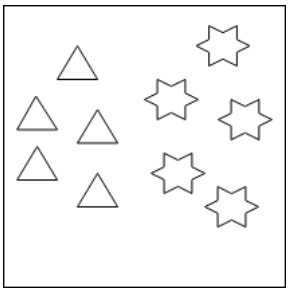
We would want our nodes to have as clear majority as possible i.e. the terminal nodes should be as homogeneous as possible.

There are different mathematical measures that quantify this homogeneity:

- Gini Index
- Entropy
- Deviance

Lets go through each of these measures of homogeneity. Lower the values of each of these measures, higher is the homogeneity of the node.

Consider the following diagram:



Gini Index: To calculate the Gini Index we use the following:

$$Gini\ Index = 1 - \sum_{i=1}^{k} p_i^2$$
$$where\ k = \ number\ of\ classes$$
$$and\ p\ = proportion\ of\ a\ class$$

Referring to the diagram above, the Gini Index will be computed as follows:

$$Gini\ Index = 1 - \Big[(\frac{5}{10})^2\ +\ (\frac{5}{10})^2\Big]\ =\ 0.5$$

Entropy: We calculate entropy as follows:

$$Entropy = -\sum_{i=1}^{k} p_i * log(p_i)$$
$$where\ k = \ number\ of\ classes$$
$$and\ p\ = proportion\ of\ a\ class$$

Referring to the diagram above, the Entropy can be computed as follows:

$$Entropy = -\Big[(\frac{5}{10}) * log(\frac{5}{10})\ +\ (\frac{5}{10}) * log(\frac{5}{10})\Big]\ =\ 0.6931472$$

Deviance: We calculate deviance as follows:

$$Deviance = -\sum_{i=1}^{k} n_i * log(p_i)$$
$$where\ k = \ number\ of\ classes$$
$$and\ p\ = proportion\ of\ a\ class$$
$$and\ n\ = no.\ of\ obs.\ in\ a\ class$$

Referring to the diagram above, we compute the deviance of that node as follows:

$$Deviance = -\Big[5 * log(\frac{5}{10})\ +\ 5 * log(\frac{5}{10})\Big]\ =\ 6.931472$$

How do these measures quantify homogeneity?

An important property of all these measures is that their value will be lowest when any of the classes have a clear majority.

In case where all the observations in a node belong to the same class, then each of the measures described above would have the value of 0.

For implementation, we can use any of the measures described above; there is no theoretical favorite.

For a detailed example of rule selection for classification, you may refer to the class presentation.

Regression:

We know that Gini Index, Entropy and Deviance are used to select rules for classification. But if the response variable is continuous, how do we select the rules?

In case of regression, the prediction is an average of the node. In order to measure how good our predictions are, we compute the error sum of squares (SSE).

$$SSE = \sum_{i=1}^{n_k} (y_i - y_k)^2$$

$$where\ n_k = \ no.\ of\ obs.\ in\ each\ node$$
$$and\ y_i = \ value\ of\ each\ obs.$$
$$and\ y_k = \ mean\ of\ the\ node$$

You may refer to the class presentation for a detailed example.

We choose the rules based on lower SSE; lower the SSE, more homogeneous is the node.

4. When do we stop splitting the nodes?

There are two natural criteria to stop splitting the nodes:

- Node is completely homogeneous - in case of classification, all observations belong to a single class and in case of regression, all the observations have the same value
- The terminal node is left with a single observation

If we let our tree grow too big, there are higher chances of over-fitting the data. It might be that the model is too perfect for the training data but does not generalize well.

In case we wish to stop growing a tree before they reach their natural stopping criterion, we can use hyper-parameters to control the size of the decision tree.

Following are some of the hyper-parameters to tune:

- max_depth: This fixes the maximum depth of the tree. If it is not set then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
- min_sample_split: The minimum number of samples required to split an internal node; default value - 2. It is a good idea to keep it slightly higher in order to reduce over-fitting of the data.
- min_sample_leaf: The minimum number of samples required to be in each leaf node on splitting their parent node. This defaults to 1. If this number is higher and a split results in a leaf node having lesser number of samples than specified then that split is cancelled.
- max_leaf_nodes: It defines the maximum number of possible leaf nodes. If it is not set then it takes an unlimited number of leaf nodes. This parameter controls the size of the tree.

Decision Tree Implementation:

We will start with building our decision tree now.

```
import pandas as pd
import numpy as np
from sklearn import tree
import numpy as np
from sklearn.metrics import roc_auc_score
import matplotlib as plt
%matplotlib inline
```

We will consider the demographic data 'census_income.csv' for this module. This is typical census data. This data has been labeled with annual income less than 50K dollars or not. We want to build a model such that given these census characteristics we can figure out if someone will fall in a category in which their income is higher than 50K dollars or not. Such models are mainly used when formulating government policies.

```
# data prep from previous module
file=r'/Users/anjal/Dropbox/0.0 Data/census_income.csv'
ci_train=pd.read_csv(file)
```

```
ci_train.head()
# display in pdf will be truncated on the right hand side
```

	age	workclass	fnlwgt	education	education.num	marital.status	occupation	relationship
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife

Lets start with the data preparation steps.

We know that there should not be any redundancy in the data. e.g. consider the 'education' and 'education_num' variables.

```
pd.crosstab(ci_train['education'],ci_train['education.num'])
# display will be truncated in pdf on the right hand side
```

education.num	1	2	3	4	5	6	7	8	9	10	11	12	13	14
education														
10th	0	0	0	0	0	933	0	0	0	0	0	0	0	0
11th	0	0	0	0	0	0	1175	0	0	0	0	0	0	0
12th	0	0	0	0	0	0	0	433	0	0	0	0	0	0
1st-4th	0	168	0	0	0	0	0	0	0	0	0	0	0	0
5th-6th	0	0	333	0	0	0	0	0	0	0	0	0	0	0
7th-8th	0	0	0	646	0	0	0	0	0	0	0	0	0	0
9th	0	0	0	0	514	0	0	0	0	0	0	0	0	0
Assoc-acdm	0	0	0	0	0	0	0	0	0	0	0	1067	0	0
Assoc-voc	0	0	0	0	0	0	0	0	0	0	1382	0	0	0
Bachelors	0	0	0	0	0	0	0	0	0	0	0	0	5355	0
Doctorate	0	0	0	0	0	0	0	0	0	0	0	0	0	0
HS-grad	0	0	0	0	0	0	0	0	10501	0	0	0	0	0
Masters	0	0	0	0	0	0	0	0	0	0	0	0	0	17
Preschool	51	0	0	0	0	0	0	0	0	0	0	0	0	0
Prof-school	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Some-college	0	0	0	0	0	0	0	0	0	7291	0	0	0	0

We observe that there is one to one correspondence. e.g. for 'education' 10th has been labelled as 'education.num' 6, 11th has been labelled as 'education.num' 7 etc. Hence, instead of using the variable 'education' we can use 'education.num' only. We will go ahead and drop 'education' here.

```
ci_train.drop(['education'],axis=1,inplace=True)
```

Now lets have a look at our outcome variable 'Y'.

```
ci_train['Y'].value_counts().index
```

```
Index([' <=50K', ' >50K'], dtype='object')
```

Notice that it has white space values which should be removed. During data preparation we need to be careful with this else when comparing these values we may get unexpected results if the white space is not considered.

Now we convert the response column 'Y' to 1's and 0's.

```
ci_train['Y']=(ci_train['Y']==' >50K').astype(int)
```

For the remaining categorical columns we will need to make dummies.

```
cat_cols=ci_train.select_dtypes(['object']).columns
cat_cols
```

```
Index(['workclass', 'marital.status', 'occupation', 'relationship', 'race',
 'sex', 'native.country'],
dtype='object')
```

When making dummies, we will ignore those values which have a frequency less than 500. You can always reduce this number and check if more dummies result in a better model.

```
for col in cat_cols:
    freqs=ci_train[col].value_counts()
    k=freqs.index[freqs>500][:-1]
    for cat in k:
        name=col+'_'+cat
        ci_train[name]=(ci_train[col]==cat).astype(int)
    del ci_train[col]
    print(col)

```

```
workclass
marital.status
occupation
relationship
race
sex
native.country
```

The above steps result in my data having 32561 rows and 39 columns including the response variable.

```
ci_train.shape
```

```
(32561, 39)
```

Next we will need to check for missing values.

```
ci_train.isnull().sum().sum()
```

```
0
```

As we can observe, there are no missing values in the data.

Next, we separate the predictors and the response variable.

```
x_train=ci_train.drop(['Y'],1)
y_train=ci_train['Y']
```

The data preparation steps are similar for all models. Here we are given only the training dataset. However, if we are given a test dataset also, we should do the data preparation steps for both training and test datasets.

Hyper Parameters For Decision Trees

- criterion: Used to set which homogeneity measure to use. Two options available: "entropy" and "gini" (default).
- max_depth: This fixes the maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Ignored if 'max_leaf_nodes' is not None.
- min_sample_split: The minimum number of samples required to split an internal node; default value - 2. It is a good idea to keep it slightly higher in order to reduce over-fitting of the data. Recommended values are between 5 to 10.
- min_sample_leaf: The minimum number of samples required to be in each leaf node on splitting their parent node. This defaults to 1. If this number is higher and a split results in a leaf node having lesser number of samples than specified then that split is cancelled.
- max_leaf_nodes: It defines the maximum number of possible leaf nodes. If None then it takes an unlimited number of leaf nodes. By default, it takes ???None??? value. This parameter controls the size of the tree. We will be finding optimal value of this through cross validation.
- class_weight: Default is None, in which case each class is given equal weight-age. If the goal of the problem is good classification instead of accuracy (especially in the case of imbalanced datasets) then you should set this to "balanced", in which case class weights assigned are inversely proportional to class frequencies in the input data.
- random_state: Used to reproduce random result.

For selecting the best parameters we will use RandomizedSearchCV instead of GridSearchCV.

Why should we consider using Randomized Search instead of Grid Search?

The variable 'params' below consists of 5 different parameters we intend to tune. Each of these parameters contain some values. The possible combinations will be 960 as shown below. If we use grid search and use a 10 fold CV, we will build around 9600 individual trees. It will result in the best possible combination but will also take a lot of time. In order to handle this, instead of trying out all 960 combinations, we can try only 10% of these combinations i.e 96 combinations. Randomized Search will randomly select 96 of the combinations and will result in good enough results - though we do not have a guarantee of the best combination. It will give us a good enough combination at a fraction of the runtime. The trade-off is between the run time and how good our result will be.

We can make the Randomized Search better by running it multiple times (each time we get a different combination) and check whether the combinations are consistent across different runs. We can expand in the neighborhood values as well e.g we get max_depth as 70 using Randomized Search; in the next run we would want to add 80 and 90 and check again. Another example would be if we get max_depth as 30, but we did not consider any other value between 30 and 50. We may want to add a max_depth of 40 and try again.

Once we get out best max_depth value as 30, we can also try values around 30, like 25, 26, 31, 32 etc which may result in better performance of the model.

```
1 2*2*8*6*5
```

```
960
```

```
# RandomSearchCV/GridSearchCV accept parameters values as dictionaries.
# In example given below we have constructed dictionary for different parameter values that we want to
# try for decision tree model
params={ 'class_weight':[None,'balanced'],
         'criterion':['entropy','gini'],
         'max_depth':[None,5,10,15,20,30,50,70],
         'min_samples_leaf':[1,2,5,10,15,20],
         'min_samples_split':[2,5,10,15,20]
       }
```

Now lets see how RandomizedSearchCV works. It works just like GridSearchCV except that we need to tell RandomizedSearchCV that out of all these combinations how many do we want to try out. e.g. We may mention that out of 960 combinations only try out 10. We use the argument n_iter to set this. Ideally, we should try about 10 to 20% of all the combinations.

```
from sklearn.model_selection import RandomizedSearchCV
```

The classifier we want to try to do this classification is the Decision Tree Classifier.

```
from sklearn.tree import DecisionTreeClassifier
```

```
clf=DecisionTreeClassifier()
```

```
# We try the decision tree classifier here supplying different parameter ranges to our randomSearchCV which in turn will pass it on to this classifier
# n_iter parameter - this number should be 10 to 20% of the total number of combinations
# n_iter parameter of RandomizedSeacrhCV controls how many parameter combinations of all given combinations will be tried

random_search=RandomizedSearchCV(clf, cv=10, param_distributions=params,
                                         scoring='roc_auc',n_iter=10, n_jobs=-1, verbose=False)
```

- In the code above, we use the object 'clf' of the DecisionTreeClassifier.
- cv=10 refers to using a 10 fold cross validation.
- We use 'scoring='roc_auc'' since its a binary classification problem. In case of multi-class classification, we would use 'accuracy' instead.
- n_iter=10 states that we will be considering 10 parameter combinations.
- n_jobs=-1 indicates that the code can use all the available CPU cores present in the system, helping us take advantage of multicore processing.
- verbose=False is used to display detailed logging information. `False` here suppresses any display. You can use integer values to display . Higher the integer value, more information is displayed while model fitting. `20` is a good number to use. Has nothing to do with model building as such, is used just for display purposes

```
random_search.fit(x_train,y_train)
```

In the output above, we can see all the parameters that were tried out.

We get the best estimator as follows:

```
random_search.best_estimator_
```

```
DecisionTreeClassifier(class_weight='balanced', criterion='entropy',
            max_depth=10, max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=10, min_samples_split=15,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')
```

The report function below gives the rank-wise details of each model.

```
# Utility function to report best scores. This simply accepts grid scores from our randomSearchCV/GridSearchCV and picks
# and gives top few combination according to their scores.

def report(results, n_top=3):
    for i in range(1, n_top + 1):
        # np.flatnonzero extracts index of `True` in a boolean array
        candidate = np.flatnonzero(results['rank_test_score'] == i)[0]
        # print rank of the model
        # values passed to function format here are put in the curly brackets when printing
        # 0 , 1 etc refer to placeholder for position of values passed to format function
        # .3f means upto 2 decimal digits
        print("Model with rank: {0}".format(i))
        # this prints cross validated performance and its standard deviation
        print("Mean validation score: {0:.5f} (std: {1:.5f})".format(
              results['mean_test_score'][candidate],
              results['std_test_score'][candidate]))
        # prints the paramter combination for which this performance was obtained
        print("Parameters: {0}".format(results['params'][candidate]))
        print("")
```

Below we can see the details of the top 5 models. The best model has the auc_roc score of 0.894 and we can check the parameters that resulted in this.

```
report(random_search.cv_results_,5)
```

> Model with rank: 1
> Mean validation score: 0.89473 (std: 0.00684)
> Parameters: {'min_samples_split': 10, 'min_samples_leaf': 15, 'max_depth': 15, 'criterion': 'gini', 'class_weight': 'balanced'}

> Model with rank: 2
> Mean validation score: 0.89388 (std: 0.00668)
> Parameters: {'min_samples_split': 20, 'min_samples_leaf': 20, 'max_depth': 30, 'criterion': 'gini', 'class_weight': None}

> Model with rank: 3
> Mean validation score: 0.89069 (std: 0.00572)
> Parameters: {'min_samples_split': 20, 'min_samples_leaf': 10, 'max_depth': 5, 'criterion': 'gini', 'class_weight': 'balanced'}

> Model with rank: 4
> Mean validation score: 0.88994 (std: 0.00735)
> Parameters: {'min_samples_split': 15, 'min_samples_leaf': 15, 'max_depth': 20, 'criterion': 'gini', 'class_weight': 'balanced'}

> Model with rank: 5
> Mean validation score: 0.88210 (std: 0.00643)
> Parameters: {'min_samples_split': 20, 'min_samples_leaf': 1, 'max_depth': 15, 'criterion': 'gini', 'class_weight': None}

We will now fit the best estimator separately.

```
dtree=random_search.best_estimator_
dtree
```

```
DecisionTreeClassifier(class_weight='balanced', criterion='entropy',
    max_depth=10, max_features=None, max_leaf_nodes=None,
    min_impurity_decrease=0.0, min_impurity_split=None,
    min_samples_leaf=10, min_samples_split=15,
    min_weight_fraction_leaf=0.0, presort=False, random_state=None,
    splitter='best')
```

```
dtree.fit(x_train,y_train)
```

The tentative performance of the model is 0.894. We can now use this model to make predictions on the test data using either the predict() or predict_proba() functions.

In order to visualize the decision tree we need to do the following:

```
dotfile = open("mytree.dot", 'w')
tree.export_graphviz(dtree,out_file=dotfile, feature_names=x_train.columns,
class_names=["0","1"], proportion=True)
dotfile.close()
```

We first need to output the decision tree to a dot file using the export_graphviz() function. 'dtree' is the model we built, 'out_file' is where we will write this decision tree, 'feature_names' are names of the features on which the rules are based, 'class_names' stores the classes and 'proportions' set to True means that we want to see the proportions in the nodes too.

Open mytree.dot file in a simple text editor and copy and paste the code at http://webgraphviz.com to visualize the tree.

We discussed only about Classification using Decision Trees.

As far as Decision Trees for Regression are concerned there will be the following differences: we use a DecisionTreeRegressor instead of DecisionTreeClassifier to create the Decision Tree object. The arguments: "class_weight" and "criterion" will not make sense in case of Regression. Also, when using Regression, the evaluation criterion will be 'neg_mean_absolute_error' instead of 'roc_auc' score.

Rest of the process remains identical for Decision Tree Regressor and Decision Tree Classifier.

Next we will discuss the issues with Decision Trees and how is the issue handled.

Random Forests

Issues with decision trees: Decision trees help capture niche non-linear patterns in data. The model may fit the training data very well but may not do well with newer data. Whenever we build a model, we want it to do well with future data too and hence needs to be as generalizable as possible. How do we handle this? On one hand we want to capture decision tree's amazing ability to capture non-linear data and at the same time we do not want it to be susceptible to noise or niche patterns from the training data.

A very simple and powerful idea resulted in the popular algorithm called Random Forest.

Random Forests introduce two levels of randomness in the tree building process which help in handling the noise in the data.

- Random Forest takes a random subset of all the observations present in training dataset
- It also takes a random subset of all the variables from the training dataset

In order to understand this idea, lets assume that we have 10000 observations and 200 variables in the dataset. In Random Forest, many decision trees are built instead of a single tree; for our example lets consider 500 trees are built. Now, if we build these trees using the same 10000 observations 500 times we will end up with the same 500 trees. But this is not what we want since we will get the same outcome from each of these 500 trees. Hence, each tree in the Random Forest does not use the entire data, but instead the individual trees are built on random subset of the observations (we can consider 10000 sample observations with replacement or fewer sample observations without replacement). In short, in the first level of randomness, instead of using the entire data, we use a random sample of the data. How does this help in cancelling out the effect of noise/niche patterns specific to the training data? By definition, noise will be a small chuck of the data. Hence, when we sample observations from the training dataset repeatedly for our 500 trees, we can safely assume that a majority of the trees built will not be affected by noise.

Now, how does taking a random subset of all the variables from the dataset help? When building a single decision tree (not in Random Forest), in order to choose a rule to split a node, the decision tree algorithm considers all the variables i.e. in our example all 200 variables will be considered. In Random Forest algorithm, on the other hand, in order to choose a rule to split a node, instead of considering all possible variables, the algorithm considers only a random subset of variables. Lets say only 20 variables are randomly picked up from the 200 variables present and only the rules generated from these 20 variables are considered when choosing the best rule to split the node. At every node a fresh random subset of variables is selected from which the best rule is picked up. How does this help in cancelling out the effect of noisy variables? The noisy variables will be a small chuck of all the variables present and when the variables are randomly selected, the chances of the noisy variables being selected are low. This in turn will not always let the noisy variables affect all the 500 individual trees made. The noisy variables may affect some of the trees a lot, but since we will take a majority over 500 trees, their effect will be minimized.

In short, the first randomness removes the effect of noisy observations and the second randomness removes the effect of noisy variables.

The final predictions made by Random Forest model will be a majority vote from all the trees in the forest in case of classification. For Regression, the predictions will be the average of predictions made by the 500 trees.

Implementation of Random Forest

Now we will build a Random Forest model. We will use the same hyper-parameters as for decision trees amongst others since Random Forest is ultimately a collection of decision trees.

Additional hyperparameters for Random Forests

- n_estimators: Number of trees to be built in the forest - default: 10; good starting point can be 100.
- max_features: Number of features being considered for selecting the best rule at each split. Note: the value of this parameter should not exceed the total number of features available.
- bootstrap : Allows for sampling with replacement or without replacement. Takes a Boolean value; if True, sampling with replacement and if False, sampling without replacement i.e sampled only once.

We will import the RandomForestClassifier as we used the DecisionTreeClassifier earlier.

```
from sklearn.ensemble import RandomForestClassifier
```

```
clf = RandomForestClassifier()
```

The dictionary below has different values of parameters that will be tried to figure out the model giving the best performance. Apart from 'n_estimators', 'max_features' and 'bootstrap' parameters which are specific to Random Forest, the rest of the parameters are the same as the ones used for decision trees.

```
# RandomSearchCV/GridSearchCV accept parameters values as dictionaries.
# In example given below we have constructed dictionary for different parameter values that we want to
# try for Random Forest model
param_dist = {"n_estimators":[100,200,300,500,700,1000],
              "max_features": [5,10,20,25,30,35],
              "bootstrap": [True, False],
              "class_weight":[None,'balanced'],
              "criterion":['entropy','gini'],
              "max_depth":[None,5,10,15,20,30,50,70],
              "min_samples_leaf":[1,2,5,10,15,20],
              "min_samples_split":[2,5,10,15,20]}
```

The number of possible combinations are huge now as shown below.

```
960*6*6*2
```

> 69120

We are looking at 69120 combinations. Having said this, each Random Forest can have around 100 to 1000 trees; on an average around 300 trees. If we try all these combinations, we are looking at building about 20 million decision trees. The time required to execute this will be huge and the performance improvement we get may not be worth the investment.

Hence, instead of looking at all the possible combinations, we will consider a random subset which will give us a good enough solution, maybe not the best; the trade off being how good the model is to the execution time.

```
960*6*6*2*300
```

> 20736000

A good number of start with would be about 10 to 20 percent of the total number of combinations.

Note: the value of max_features cannot exceed the total number of features in the data. As seen below, max_features cannot exceed 38.

```
x_train.shape
```

> (32561, 38)

```
n_iter_search = 10 # this number should be 10 to 20% of the total number of combinations
#n_iter parameter of RandomizedSeacrhCV controls, how many parameter combination will be tried; out of all possible given values

# We try the random forest classifier here supplying different parameter ranges to our RandomizedSearchCV which in turn
# will pass it on to this classifier
random_search = RandomizedSearchCV(clf, param_distributions=param_dist,
                                   n_iter=n_iter_search,
                                   scoring='roc_auc',
                                   cv=5, n_jobs=-1,verbose=False)
random_search.fit(x_train, y_train)
```

Amongst all the models built, we get the best estimator using the 'best_estimator_' argument of the random_search object.

```
random_search.best_estimator_
```

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='entropy',
    max_depth=10, max_features=20, max_leaf_nodes=None,
    min_impurity_decrease=0.0, min_impurity_split=None,
    min_samples_leaf=1, min_samples_split=2,
    min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=None,
    oob_score=False, random_state=None, verbose=0,
    warm_start=False)
```

Note: This is a result from one of the runs, you can very well get different results from a different run. Your results need not match with this.

Looking at the outcome of the report function, we observe that the model with rank 1 has the mean validation score of 0.918 which is a slight improvement over decision trees. Trying higher values of 'cv' and 'n_iter' arguments should give better models.

```
report(random_search.cv_results_,5)
```

```
Model with rank: 1
Mean validation score: 0.91790 (std: 0.00296)
Parameters: {'n_estimators': 100, 'min_samples_split': 15, 'min_samples_leaf': 2, 'max_features': 5, 'max_depth': 30, 'criterion': 'entropy', 'class_weight': 'balanced', 'bootstrap': True}
```

```
Model with rank: 2
Mean validation score: 0.91643 (std: 0.00334)
Parameters: {'n_estimators': 500, 'min_samples_split': 10, 'min_samples_leaf': 10, 'max_features': 20, 'max_depth': 10, 'criterion': 'gini', 'class_weight': 'balanced', 'bootstrap': False}
```

```
Model with rank: 3
Mean validation score: 0.91434 (std: 0.00365)
Parameters: {'n_estimators': 300, 'min_samples_split': 10, 'min_samples_leaf': 5, 'max_features': 25, 'max_depth': 30, 'criterion': 'gini', 'class_weight': None, 'bootstrap': True}
```

```
Model with rank: 4
Mean validation score: 0.91338 (std: 0.00333)
Parameters: {'n_estimators': 700, 'min_samples_split': 20, 'min_samples_leaf': 20, 'max_features': 25, 'max_depth': 30, 'criterion': 'entropy', 'class_weight': 'balanced', 'bootstrap': False}
```

```
Model with rank: 5
Mean validation score: 0.91172 (std: 0.00273)
Parameters: {'n_estimators': 200, 'min_samples_split': 15, 'min_samples_leaf': 5, 'max_features': 30, 'max_depth': 15, 'criterion': 'gini', 'class_weight': 'balanced', 'bootstrap': False}
```

Now, lets build the model giving the best performance.

```
rf = RandomForestClassifier(bootstrap=True, class_weight=None, criterion='entropy',
            max_depth=10, max_features=20, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=None,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```

```
rf.fit(x_train,y_train)
```

Once the model is fit, we can use it to make predictions using hard classes or probability.

Feature Importance

In scikit-learn, one of the ways in which feature importance is described is by something called as "mean decrease impurity". This "mean decrease impurity" is the total decrease in node impurity averaged over all trees built in the random forest.

If there is a big decrease in impurity, then the feature is important else it is not important.

In the code below, feature importance can be obtained using the 'feature_importances_' attribute of the model. The use of this attribute makes sense only after the model is fit. We store the feature importance along with the corresponding feature names in a dataframe and then sort them according to importance.

```
feat_imp_df=pd.DataFrame({'features':x_train.columns,'importance':rf.feature_import
ances_})

print(feat_imp_df.sort_values('importance',ascending=False).head())

print(feat_imp_df.sort_values('importance',ascending=False).tail())
```

```
??? features                              importance
12 marital.status_ Married-civ-spouse   0.240887
3   capital.gain                       0.189728
2   education.num                      0.162019
28  relationship_ Husband              0.085335
0   age                                0.075243


???       features                          importance
15   marital.status_ Separated             0.000524
37   native.country_ Mexico               0.000482
23   occupation_ ?                        0.000426
25   occupation_ Handlers-cleaners         0.000389
31   relationship_ Unmarried               0.000365
```

We can observe that 'marital.status_ Married-civ-spouse' is identified as the most important variable and 'relationship_ Unmarried' as the least important.

Random Forests can also be used as a dimensionality reduction technique. In case of high dimensional data, we can run a random forest model and get features sorted by importance. We can then go ahead and choose the top, say 100, features for further processing. Using this technique we would not be losing relevant information and at the same time we can drastically reduce the number of features considered. In other words, even if we do not use the Random Forest model as the final model, we can use it for reducing the features.

Partial Dependence Plot

Random Forest model is an example of a black-box model - where we do not get any coefficients for any variable and hence are unable to make an interpretation of different variables on the response variable.

In order to make an interpretation of a variable used in Random Forest, we need to make prediction on the entire data and average it for the variable we are interested in.

Lets take an example of the 'education.num' variable.

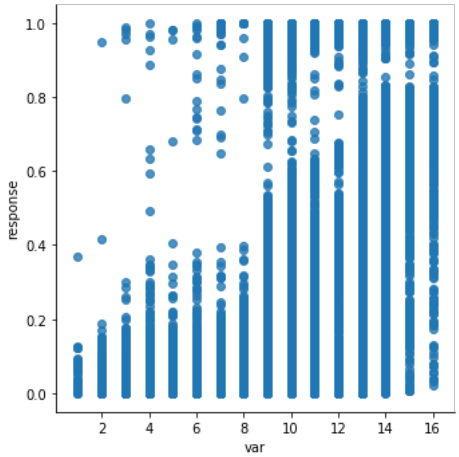
```
var_name='education.num'
preds=rf.predict_proba(x_train)[:,1] # making a prediction in the entire data
```

```
var_data=pd.DataFrame({'var':x_train[var_name],'response':preds})
var_data.head() # Create a dataframe of the 'education.num' variable and the
corresponding predictions
```

	var	response
0	13	0.079281
1	13	0.285775
2	9	0.027492
3	7	0.114180
4	13	0.506204

We plot the two columns 'education.num' against the 'response' as shown below but it is not very informative i.e. there is a lot of variation since the response contains the effects of other variables also.

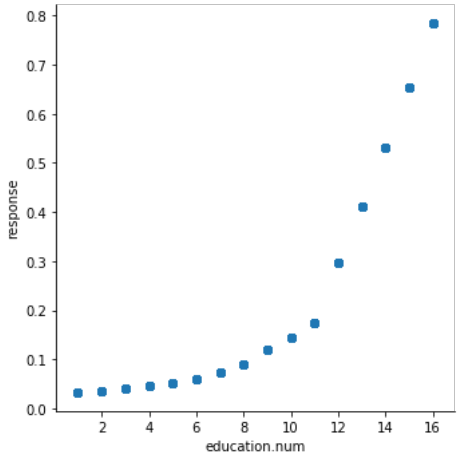
```
import seaborn as sns
sns.lmplot(x='var',y='response',data=var_data,fit_reg=False)
```



We will plot a smoothing curve through this which will give an approximate effect of the variable 'education.num' on the response. We basically will average the response at each of the 'education.num' values.

```
import statsmodels.api as sm
smooth_data=sm.nonparametric.lowess(var_data['response'],var_data['var'])
```

```
df=pd.DataFrame({'response':smooth_data[:,1],var_name:smooth_data[:,0]})
sns.lmplot(x=var_name,y='response',data=df,fit_reg=False)
```



In the plot above we notice that as the 'education.num' value goes up the chances of having income greater than 50000 dollars go up. The 'education.num' variable does not have much impact on the response till its value is around 10 or 12, but then it rises steeply indicating that with higher education the probability of earning more than 50000 dollars increases.

This exercise can be done for any of the other variables we wish to assess. However, it does not make sense for dummy variables since it has only two values. But it works fine for continuous variables.

Chapter 6 : Boosting Machines

In previous module we saw models based on bagging; random forests and extraTrees where each individual model was independent and eventual prediction of the ensemble of these models was a simple majority vote or average depending on whether the problem was of classification of regression.

The randomness in the process, helped the model become less affected by noise and more generalizable. However this bagging did not really help in underlying models to become better at extracting more complex patterns.

Boosting machines go that extra step, in modern implementation, both the ideas; using randomness to make models generalizable and boosting[which we'll study in few moments] are used . You can consider boosting machines to be more powerful than bagging models . However that comes with downside of them being prone to over-fitting in theory. We'll learn about Extreme Gradient Boosting (Xgboost) which goes one step further and adds the element of regularization to boosting machines and some other radical changes to become one of the most successful Machine Learning Algorithms in the recent history.

Just like bagging , boosting machines also are made up of multiple individual models . However in boosting machines , individual models are built sequentially and eventual model is summation of individual models not the average . Formally for a boosting machine, our $F(X_i)$ or more formally with `t` individual models , $F_t(X_i)$ is written as :

$$F_t(X_i) = f_1(X_i) + f_2(X_i) \ldots\ldots\ldots\ldots + f_t(X_i)$$

where $f_t(X_i)$ represents individual models . As mentioned earlier , these individual models are built sequentially. Patterns which could not be captured by the model so far become target for the next guy. Its fairly intuitive to understand in context of regression model [The one with the continuous numeric target] .

for f_1 , target is simply the original outcome y_i, but as we go forward , the target simply becomes errors remaining so far :

$$f_1 \rightarrow y_i$$

$$f_2 \rightarrow (y_i - f_1)$$

$$f_3 \rightarrow (y_i - f_1 - f_2)$$

$$\vdots$$

$$f_{t+1} \rightarrow (y_i - F_t)$$

by the looks of it this looks like a recipe for disaster , certainly over-fitting . Remember that we are trying to reduce the error here on training data, and if we keep on fitting models on the residuals , eventually we'll start to severely over-fit.

Why Weak-Learners

In order to avoid the over-fitting , we can chose our individual models to be weak-learners. Models which are incapable of over-fitting themselves. In fact the ones which are not good models taken individually. Individually they are only capable of capturing most strong and hence reliable patterns. And upon boosting , such consistent patterns extracted (though with changing target for each) taken together make for a very strong and yet somewhat robust to over-fitting model.

What weak-learners

In theory, any kind of base model can be made a weak learner. Few examples :

- Linear Regression which makes use of say only $(\frac{1}{10})^{th}$ of the variables at each step

- Linear Regression with very very high penalty [Large value of λ in L1/L2 regularization]
- Tree Stumps : Very Shallow Decision Trees [low depth]

In Practice however , `Tree Stumps` are popular and you will find them implemented almost everywhere.

Remember that decision trees start to over-fit [extract niche patterns from the training data] when we grow them too large and start to pick partition rules from smaller and smaller chunk of the training data. Shallow decision trees do not reach to that point and hence are incapable of individually over-fitting .

Gradient Boosting Machines

So far we haven't formally discussed how do we go about building this so called boosted model. Although the example taken above in context of regression models, seem like a no-brainer. However, that's only good for building intuition in the beginning. It fails to generalize pretty fast with slight changes in the cost functions . Gradient Boosting Machines combine the idea of `Gradient Descent` and `Boosting` to give a generic method which works with any cost/loss formulation.

Lets see how do we transition from Gradient Decent for parameters to Gradient Boosting Machines . If you recall, idea behind gradient decent was that in each iteration we update the parameters by changing [updating] them like this :

$$\beta \rightarrow \beta - \eta * \nabla C$$

or

$$\Delta\beta = -\eta * \nabla C$$

- Parameters βs are getting updated and the gradient of the cost is taken w.r.t. to parameters
- In context of boosting machines however , the Model itself is getting updated. in every iteration $F_{t+1} = F_t + f_{t+1}$., model is updated by f_{t+1} . This update, using the `Gradient Descent` will be equal to $-\eta * \nabla C$, however the gradient here will be taken w.r.t. to what is being updated that is F_t.
- Before we are able to take the derivative/gradient of the loss w.r.t. F_t, we'll need to write the cost/loss in terms of F_t
- f_{t+1} being equal to $-\eta * \nabla C$ doesn't intuitively make sense . f_{t+1} is after all a model [Tree stump]. What does it mean for a model to be `equal` to something. It means that while we are building the model we'll take $-\eta * \nabla C$ as the target for the model .

Formally if we write our cost/loss =$\sum C(y_i, F_t)$

where

$$F_t = f_1 + f_2 + \cdots + f_t$$

then

$$f_{t+1} \rightarrow -\eta * \frac{\delta C}{\delta F_t} \qquad \cdots (1)$$

GBM for Regression

for regression if you recall our traditional loss is nothing but squared error

$$C = (y_i - F_t)^2$$

if we take the derivation of this w.r.t. F_t, we get :

$$\frac{\delta C}{\delta F_t} = -2 * (y_i - F_t)$$

putting this back in $\cdots (1)$, we get

$$f_{t+1} \rightarrow \eta * (y_i - F_t)$$

This simply means that every next shallow decision tree will take small fraction of the remaining error as its target. η is there to ensure that any one individual model doesn't end up contributing too heavily towards the eventual prediction.

GBM for Classification

If you revisit your `Intro to ML` chapter and look up the discussion on cost/loss for classification, you'll find this formulation for loss

$$C = -[y_i * F_t - log(1 + e^{F_t})]$$

lets take its derivative w.r.t. F_t

$$\frac{\delta C}{\delta F_t} = -[y_i - \frac{e^{F_t}}{1 + e^{F_t}}]$$

$$= -[y_i - \frac{1}{1 + e^{-F_t}}]$$

in the same section you would have found that probability p_i is actually represented as this :

$$p_i = \frac{1}{1 + e^{-F_t}}$$

using the same we can write :

$$\frac{\delta C}{\delta F_t} = -[y_i - p_i]$$

putting this back in $\cdots$ (1) , we get

$$f_{t+1} \rightarrow \eta * (y_i - p_i)$$

Couple things that you need to realize about gbm for classification :

- Each individual shallow tree is a regression tree [not a classification tree] as the target for them is a continuous numeric number
- p_i is not a simple summation of probability scores given by individual shallow tree models. No.
- $F_{t+1} = F_t + f_{t+1}$ and $p_{t+1} = \frac{1}{1+e^{-(F_t + f_{t+1})}}$

You should realize that this process can be done for any alternate cost/loss formulation as well. All that we need to do is that to write the cost w.r.t. model itself and then define its derivative.

Next we look at parameters to tune in GBM

GBM parameters

- n_estimators [default = 100]:

Number of boosted individual models . there is no upper limit.

- learning_rate [default=0.1]

learning rate shrinks the contribution of each tree by `learning_rate`.There is a trade-off between learning_rate and n_estimators. A small learning rate will require large number of n_estimator.

- max_depth [default=3]

depth of the individual tree models . High number here will lead to complex/over-fit model

- min_samples_split[default=2]

The minimum number of samples required to split an internal node:

```
- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and
  `ceil(min_samples_split * n_samples)` are the minimum
  number of samples for each split.
```

- min_samples_leaf[default = 1]

The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model,
especially in regression.

```
- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and
  `ceil(min_samples_leaf * n_samples)` are the minimum
  number of samples for each node.
```

Note : Both `min_samples_leaf` and `min_samples_split` however might be rendered useless as we keep max_depth low to ensure individual models remaining weak learners. The situation for all practical purposes might never arise where these two become relevant due to shallow trees

- subsample [default =1]

The fraction of samples to be used for fitting the individual base learners. If smaller than 1.0 this results in Stochastic Gradient Boosting.

- max_features [default = None]

The number of features to consider when looking for the best split:

```
- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and
  `int(max_features * n_features)` features are considered at each
  split.
- If "auto", then `max_features=sqrt(n_features)`.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.
```

Regression problem with GBM

`Problem Statement` : We are going to build a model for predicting bike sharing numbers given various weather factors and other characteristic for the days. Lets read the data

```
import pandas as pd
import numpy as np
file=r'/Users/lalitsachan/Dropbox/0.0 Data/Cycle_Shared.csv'
bike=pd.read_csv(file)
bike.head()
# display will be truncated in pdf on the right hand side
```

	instant	dteday	season	yr	mnth	holiday	weekday	workingday	weathersit	temp	a
0	1	2011-01-01	1	0	1	0	6	0	2	0.344167	0
1	2	2011-01-02	1	0	1	0	0	0	2	0.363478	0
2	3	2011-01-03	1	0	1	0	1	1	1	0.196364	0
3	4	2011-01-04	1	0	1	0	2	1	1	0.200000	0
4	5	2011-01-05	1	0	1	0	3	1	1	0.226957	0

```
bike.shape
```

(731, 16)

- instant : will be dropped because of being an id variable
- dteday : we'll extract day of month from this and drop the column
- season, mnth, weekday, weathersit : will be treated as categorical variable
- casual,registered : cnt our target variable is a simple sum of these
- yr : will be dropped as future instances will not take the same value so no point in treating this as categorical variable and it takes too few values for it to be treated like a numeric one

```
bike.drop(['yr','instant','casual','registered'],axis=1,inplace=True)
```

```
bike['dteday']=pd.to_datetime(bike['dteday'])
bike['day']=bike['dteday'].dt.month
del bike['dteday']
```

```
for col in ['season', 'mnth', 'weekday', 'weathersit']:

    k=bike[col].value_counts()
    cats=k.index[k>30][:-1]
    for cat in cats :
        name=col+'_'+str(cat)
        bike[name]=(bike[col]==cat).astype(int)

    del bike[col]

```

```
x_train=bike.drop('cnt',1)
y_train=bike['cnt']
```

```
gbm_params={'n_estimators':[50,100,200],
           'learning_rate': [0.01,.05,0.1,0.4,0.8,1],
            'max_depth':[1,2,3,4,5,6],
            'subsample':[0.5,0.8,1],
            'max_features':[5,10,15,20,28]
           }
# Note : keep in mind that this dataset is too small and ideally we should avoid complex models
```

```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import RandomizedSearchCV
```

```
model=GradientBoostingRegressor()
random_search=RandomizedSearchCV(model,scoring='neg_mean_absolute_error',
                                 param_distributions=gbm_params,
                                 cv=10,n_iter=10,
                                 n_jobs=-1,verbose=False)
```

```
random_search.fit(x_train,y_train)
```

```
def report(results, n_top=3):
    for i in range(1, n_top + 1):
        candidates = np.flatnonzero(results['rank_test_score'] ==i)
        for candidate in candidates:
            print("Model with rank: {0}".format(i))
            print("Mean validation score: {0:.5f} (std: {1:.5f})".format(
                  results['mean_test_score'][candidate],
                  results['std_test_score'][candidate]))
            print("Parameters: {0}".format(results['params'][candidate]))
            print("")
```

```
report(random_search.cv_results_,3)
```

```
Model with rank: 1
Mean validation score: -1569.07622 (std: 664.47666)
Parameters: {'subsample': 0.8, 'n_estimators': 50, 'max_features': 28, 'max_depth': 5,
'learning_rate': 0.01}
```

```
Model with rank: 2
Mean validation score: -1645.59695 (std: 197.54250)
Parameters: {'subsample': 1, 'n_estimators': 50, 'max_features': 20, 'max_depth': 3,
'learning_rate': 0.05}
```

```
Model with rank: 3
Mean validation score: -1658.20089 (std: 169.47697)
Parameters: {'subsample': 0.5, 'n_estimators': 100, 'max_features': 10, 'max_depth': 2,
'learning_rate': 0.05}
```

This gives us our best model parameters as well as its performance . We can further extract variable importance and make partial dependence plots here also as we did for randomForest results in the earlier module

Classification problem with GBM

`Problem Statement` : Given the demographic details , we'll a model to predict whether a person's income is greater than $50K or not

```
# many imports will not be explicitly done now onwards here
# because they have already happened earlier in this chapter
# if you are working on this in a fresh notebook
# do include those imports
file = r'/Users/lalitsachan/Dropbox/0.0 Data/census_income.csv'
cd=pd.read_csv(file)
cd.head()
# display will be truncated in pdf on the right hand side
```

	age	workclass	fnlwgt	education	education.num	marital.status	occupation	relationship
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife

```
cd.shape
```

(32561, 15)

- workclass,marital.status , occupation, relationship, race,sex, native.country : will create dummies
- education: will dropped because it has direct correspondence with education.num [check with pd.crosstab]
- Y : target variable , will be converted to 1/0

```
cd.drop('education',1,inplace=True)
```

```
cd['Y'].unique()
```

array([' <=50K', ' >50K'], dtype=object)

```
cd['Y']=(cd['Y']==' >50K').astype(int)
for col in ['workclass','marital.status' , 'occupation',
            'relationship', 'race','sex', 'native.country']:
    k=cd[col].value_counts()
    cats=k.index[k>300][:-1]
    for cat in cats :
        name=col+'_'+str(cat)
        cd[name]=(cd[col]==cat).astype(int)

    del cd[col]
```

```
cd.shape
```

(32561, 41)

```
x_train=cd.drop('Y',1)
y_train=cd['Y']

```

```
gbm_params={'n_estimators':[50,100,200,500],
            'learning_rate': [0.01,.05,0.1,0.4,0.8,1],
             'max_depth':[1,2,3,4,5,6],
             'subsample':[0.5,0.8,1],
             'max_features':[0.1,0.3,0.5,0.8,1]
            }
```

```
from sklearn.ensemble import GradientBoostingClassifier
```

```
model=GradientBoostingClassifier()
random_search=RandomizedSearchCV(model,scoring='roc_auc',
                                 param_distributions=gbm_params,
                                 cv=10,n_iter=10,
                                 n_jobs=-1,verbose=False)
```

```
random_search.fit(x_train,y_train)
```

```
report(random_search.cv_results_,3)
```

> Model with rank: 1
> Mean validation score: 0.92640 (std: 0.00292)
> Parameters: {'subsample': 0.8, 'n_estimators': 500, 'max_features': 0.3, 'max_depth': 4, 'learning_rate': 0.1}

> Model with rank: 2
> Mean validation score: 0.91770 (std: 0.00358)
> Parameters: {'subsample': 1, 'n_estimators': 200, 'max_features': 1, 'max_depth': 3, 'learning_rate': 0.4}

> Model with rank: 3
> Mean validation score: 0.91750 (std: 0.00400)
> Parameters: {'subsample': 1, 'n_estimators': 500, 'max_features': 0.3, 'max_depth': 4, 'learning_rate': 0.01}

This gives us our best model parameters as well as its performance . We can further extract variable importance and make partial dependence plots here also as we did for randomForest results in the earlier module

Extreme Gradient Boosting Machines (Xgboost)

Although boosting machines are very powerful algorithms to extract very complex patterns and they have been fairly popular in their glory days. However they had few issues which were needed to be addressed with few new clever ideas. Xgboost does just that. The issues that it addresses are :

- GBM relies on individual models to be weak learners , but there is no framework enforcing this; ensuring that individual models are weak learners
- Entire focus instead is on bringing down the cost, without any regularization on the complexity of the model which is a recipe for over-fitting eventually
- The contribution from individual models (scores/update) are not aligned with the idea of optimizing the cost. they are simple averages instead .

Lets see how Xgboost addresses these concerns. The discussion will be deeply mathematical and pretty complex at places. Even if it doesn't make sense in one go, don't worry about it. Focus on implementation and usage steps, you can always give more passes to understand mathematical details later on

New cost formulation with regularization

We'll be using the word `obj` short for objective function, often used in optimization as an alternate name for loss/cost . Here is the new objective for t^{th} individual model that we are trying to add to our overall model F_{t-1} so far, as an update to arrive at F_t

$$obj^{(t)} = \sum_{i=1}^{n} L(y_i, F_t) + \sum_{i=1}^{t} \Omega(f_i) \qquad \cdots (2)$$

Where the first term represents the traditional loss that we have been using so far, for GBM, the second term represents the regularization/penalty on complexity for each of the individual models

As mentioned earlier , Xgboost , along with using regularization , uses some clever modification to loss formulation that eventual make it a great algorithm. One of them is what we are going to discuss next. It is using Taylor's expansion of the traditional loss

Taylor's expansion of loss

$$g(x+a) = g(x) + ag'(x) + \frac{a^2}{2}g''(x) + \frac{a^3}{6}g^{(3)}(x) + \cdots$$

This expression written above represents , general Taylor's expansion of a function `g` where `a` is a very small quantity and g' is first order derivative , g'' is second order derivative and so on. Generally , higher order terms (terms with higher powers of `a` than a^2) are ignored , considering them be too close to zero , given `a` is a very small number . Lets Re-write (2) using this idea :

$$obj^{(t)} = \sum_{i=1}^{n} L(y_i, F_{t-1} + f_t) + \sum_{i=1}^{t} \Omega(f_i)$$

We have simply written F_t as $F_{t-1} + f_t$, here f_t is a small update , like `a` in above expression. $L(y_i, F_{t-1})$ is our `g` here in the expression

$$obj^{(t)} = \sum_{i=1}^{n} L(y_i, F_{t-1}) + f_t * \frac{\delta L(y_i, F_{t-1})}{\delta F_{t-1}} + \frac{{f_t}^2}{2} * \frac{\delta^2 L(y_i, F_{t-1})}{\delta^2 F_{t-1}} + \sum_{i=1}^{t} \Omega(f_i)$$

Notice that we have ignored higher order terms . Lets simplify this expression so that we don't have to write such complex mathematical expressions every time we write this objective function

- For an update f_t on an existing model F_{t-1}, the term $\sum_{i=1}^{n} L(y_i, F_{t-1})$ will be constant , we don't have to worry about optimizing that in the objective function. we can ignore that
- $g_i = \frac{\delta L(y_i, F_{t-1})}{\delta F_{t-1}}$
- $h_i = \frac{\delta^2 L(y_i, F_{t-1})}{\delta^2 F_{t-1}}$

Considering these ideas , our objective function can be written as follows :

$$obj^{(t)} = \sum_{i=1}^{n} [f_t * g_i + \frac{{f_t}^2}{2} * h_i] + \sum_{i=1}^{t} \Omega(f_i) \qquad \cdots (3)$$

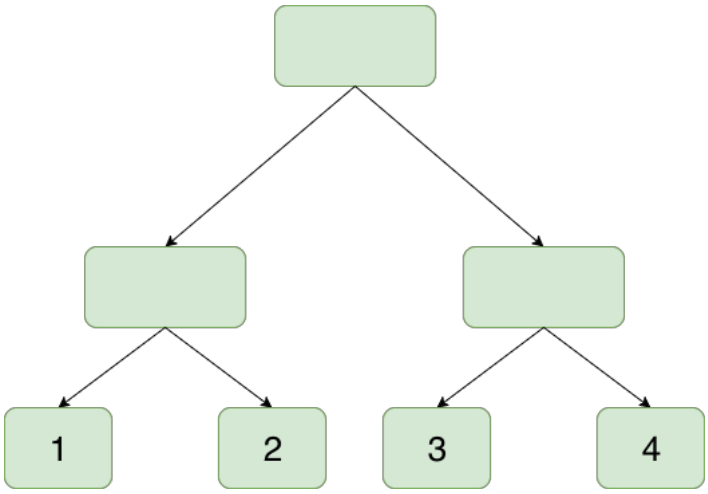
We'll see in sometime , how this helps . So far, we haven't concretely defined the penalty term.

Regularization Term

We have just written some general function Ω without really being explicit about it . Before blindly diving into how we define Ω, lets consider, what do we want to penalize here, what do we want to add the regularization term for :

- We want to ensure that individual models remain weak learners. In context of shallow decision tree, we need to ensure the tree size remains small. We can ensure that by adding penalty to the tree size (number of terminal nodes in the tree)
- We want the updates coming from individual models to be small , we can add L1/L2 penalty on the scores coming from individual trees

But what are these `scores` coming from individual trees. How do we represent a tree mathematically ? Lets have a look at what a tree is :



As we know from our discussion in the last module, every observation which ends up in the terminal node `1` will have some fixed score w_1 and so on. Lets say $q(x_i)$ represents the set of rules which tree is made of . The outcome of $q(x_i)$ for each observation x_i is one of these numbers : `{1,2,3,4}`. Essentially telling that in which terminal node the observation will end up . We can formally define our f_t or decision tree as follows :

$$f_t(x_i) = w_{q(x_i)}$$

Where $q(x_i) \in \{1, 2, 3 \cdots T\}$, given there are `T` terminal nodes in the tree

Now that we have defined the model f_t (Decision Tree in this case) , lets move on to formalize a regularization term for the same. considering the two concerns that we raised above , here is one regularization/penalty proposed by `Xgboost` team . You can always come up with some formulation of your own, this one works pretty well though.

First term penalizes size `T` of the trees thus ensuring that they are weak learners. Second term penalizes outcome/scores of trees , thus ensuring that update coming from individual model remains small . γ here controls the extent of penalty on the size of the penalty , and λ controls the extent of `L2` penalty on the scores .

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} {w_j}^2$$

Consolidating updates to cost formulation

Adding the proposed regularization to the objective function (3) that we arrived at earlier, gets us to this :

$$obj^{(t)} = \sum_{i=1}^{n} [g_i w_{q(x_i)} + \frac{1}{2} h_i {w^2}_{q(x_i)}] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} {w_j}^2 \qquad \cdots (4)$$

(4) still has some issues though. First term is written in terms of summation over all observation and second summation term is written as summation over terminal nodes of the tree. In order to consolidate things , we need to bring them under summation over same thing.

consider set I_j which contain indices `i` for observations which end up in j^{th} node . outcome of the model , for j^{th} node is already defined as w_j . Using this, we can write this all the summation terms in (4) over nodes of the tree .

$$obj^{(t)} = \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) {w_j}^2] + \gamma T$$

we'll simplify this a little by considering $G_j = \sum_{i \in I_j} g_i$ and $H_j = \sum_{i \in I_j} h_i$

$$obj^{(t)} = \sum_{j=1}^{T}[G_j w_j + \frac{1}{2}(H_j + \lambda){w_j}^2] + \gamma T \qquad \cdots (5)$$

Optimal value of model score/weights w_j

In traditional decision trees (same idea gets used in traditional `GBM` also) , score/outcome of the trees at nodes (w_j for j^{th} node) is simple average of the values in the node. However `Xgboost` team modified that idea as well. Instead of using simple average of outcomes in j^{th} node as output, they use optimal value as per the objective score in (5)

(5) is quadratic equation in w_j, it takes minimum value when :

$$w_j = -\frac{G_j}{H_j + \lambda}$$

putting this optimal value of w_j back in (5) we get :

$$obj^{(t)} = -\frac{1}{2}\sum_{j=1}^{T}\frac{{G_j}^2}{H_j + \lambda} + \gamma T \qquad \cdots (6)$$

OK, we have put a lot of work in developing/understanding this `objective function`. Where do we use this ? . This is the objective function used to select rules when we are adding new partition to our tree (essentially building the tree)

Consider that we are splitting a node into its children. Some of the observations will go to right hand side and some will go to left hand side node . Change in objective function when we create this nodes going to be :

$$\text{Gain} = \frac{1}{2}\Big[\frac{{G_L}^2}{H_L + \lambda} + \frac{{G_R}^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda}\Big] - \gamma$$

This will be evaluated for each candidate rule, and the one with highest gain will be selected as the rule for that particular node. This is another big change to how trees were being built in tradition algorithms .

Before starting with building `Xgboost` models, lets take a quick look at parameters associated with the implementation in python and what should we keep in mind when we are tuning them .

Xgboost parameters

- n_estimators [default = 100]

Number of boosted trees in the model. If learning_rate/eta is small, you'll need higher number of boosted trees to capture proper patterns in the data

- eta [default=0.1, alias: learning_rate]

Step size shrinkage used in update to prevents over-fitting. After each boosting step, we can directly get the weights of new features, and eta shrinks the feature weights to make the boosting process more conservative.
range: [0,1]

alias is nothing but an alternate name for the argument

- gamma [default=0, alias: min_split_loss]

Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger gamma is, the more conservative(less prone to Overfitting) the algorithm will be.
range: [0,???]

- max_depth [default=3]

Maximum depth of a tree. Increasing this value will make the model more complex and more likely to over-fit.

- min_child_weight [default=1]

Minimum sum of instance weight (hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than min_child_weight, then the building process will give up further partitioning. In linear regression task, this simply corresponds to minimum number of instances needed to be in each node. for classification it corresponds to minimum amount of impurity required to split a node. The larger min_child_weight is, the more conservative the algorithm will be.
range: [0,???]

- max_delta_step [default=0]

Maximum delta step we allow each leaf output to be. If the value is set to 0, it means there is no constraint. If it is set to a positive value, it can help making the update step more conservative. Usually this parameter is not needed, **but it might help in logistic regression when class is extremely imbalanced.** Set it to value of 1-10 might help control the update.
range: [0,???]

- subsample [default=1]

Subsample ratio of the training instances. Setting it to 0.5 means that XGBoost would randomly sample half of the training data prior to growing trees. and this will prevent over-fitting. Subsampling will occur once in every boosting iteration.
range: (0,1]

- colsample_bytree, colsample_bylevel, colsample_bynode [default=1]

This is a family of parameters for subsampling of columns. - All colsample_by* parameters have a range of (0, 1], the default value of 1, and specify the fraction of columns to be subsampled. `colsample_bytree` is the subsample ratio of columns when constructing each tree. Subsampling occurs once for every tree constructed.`colsample_bylevel` is the subsample ratio of columns for each level. Subsampling occurs once for every new depth level reached in a tree. Columns are subsampled from the set of columns chosen for the current tree.`colsample_bynode` is the subsample ratio of columns for each node (split). Subsampling occurs once every time a new split is evaluated. Columns are subsampled from the set of columns chosen for the current level.colsample_by* parameters work cumulatively. For instance, the combination {'colsample_bytree':0.5, 'colsample_bylevel':0.5, 'colsample_bynode':0.5} with 64 features will leave 4 features to choose from at each split.

- lambda [default=1, alias: reg_lambda]

L2 regularization term on weights. Increasing this value will make model more conservative.

- alpha [default=0, alias: reg_alpha]

L1 regularization term on weights. Increasing this value will make model more conservative.
tree_method string [default= auto]

- scale_pos_weight [default=1]

Control the balance of positive and negative weights, **useful for unbalanced classes.** A typical value to consider: $\frac{\text{Frequency of negative instances}}{\text{Frequency of positive instances}}$. Not Relevant for regression problems

Regression with Xgboost (Sequential Tuning)

If we tune all the parameters together , there are chances that our results will be much far from the best. There are many parameters where variation doesn't impact the performance too much and we can tune them later once we have fixed values of parameters with volatile performance.

As a general strategy you can start with tuning number of trees or n_estimators , in case of boosting machines , learning_rate is directly related with n_estimators . A very low learning_rate will need high number of n_estimators . We can start with a decent fixed learning rate and tune n_estimaors for it.

All can be left as default for now except subsample , colsample_bytree and colsample_bylevel, these are set to default=1, we'll take a more conservative value 0.8

```
from sklearn.model_selection import GridSearchCV
from xgboost.sklearn import XGBRegressor
xgb_params = { "n_estimators":[25,50,100,150,200] }

xgb1=XGBRegressor(subsample=0.8,colsample_bylevel=0.8,colsample_bytree=0.8)
grid_search=GridSearchCV(xgb1,cv=10,param_grid=xgb_params,scoring='neg_mean_absolute_error',verbose=False,n_jobs=-1)
x_train=bike.drop('cnt',1)
y_train=bike['cnt']
grid_search.fit(x_train,y_train)
report(grid_search.cv_results_,3)
```

> Model with rank: 1
> Mean validation score: -1633.71079 (std: 423.52205)
> Parameters: {'n_estimators': 25}

> Model with rank: 2
> Mean validation score: -1711.53315 (std: 185.46924)
> Parameters: {'n_estimators': 50}

> Model with rank: 3
> Mean validation score: -1758.83604 (std: 187.03617)
> Parameters: {'n_estimators': 100}

we'll use `n_estimators` as 25 here onwards. Lets tune other remaining parameters sequentially. Next we'll tune max_depth,gamma and min_child_weight, which control over-fit by controlling size of individual trees

```
xgb_params={
            "gamma":[0,2,5,8,10],
            "max_depth": [2,3,4,5,6,7,8],
            "min_child_weight":[0.5,1,2,5,10]
            }
xgb2=XGBRegressor(n_estimators=25,subsample=0.8,
                  colsample_bylevel=0.8,colsample_bytree=0.8)
random_search = RandomizedSearchCV( xgb2, param_distributions = xgb_params,
                                    n_iter = 20, cv= 10,
                                    scoring ='neg_mean_absolute_error',
                                    n_jobs =-1, verbose=False)
random_search.fit(x_train,y_train)
report(random_search.cv_results_,3)
```

> Model with rank: 1
> Mean validation score: -1561.90731 (std: 420.92672)
> Parameters: {'min_child_weight': 0.5, 'max_depth': 2, 'gamma': 2}

> Model with rank: 1
> Mean validation score: -1561.90731 (std: 420.92672)
> Parameters: {'min_child_weight': 0.5, 'max_depth': 2, 'gamma': 8}

> Model with rank: 3
> Mean validation score: -1633.71079 (std: 423.52205)
> Parameters: {'min_child_weight': 1, 'max_depth': 3, 'gamma': 5}

> Model with rank: 3
> Mean validation score: -1633.71079 (std: 423.52205)
> Parameters: {'min_child_weight': 1, 'max_depth': 3, 'gamma': 8}

Next we'll tune subsampling arguments to take care of potential effect of noisy observations [if this was classification problem , we'd have tuned `max_delta_step` and `scale_pos_weight` to take care of impact of class imbalance in the data before doing this]. Notice that now we'll use the values of parameters tuned so far from the best model outcomes from the gridsearch .\

```
xgb_params={
            'subsample':[i/10 for i in range(5,11)],
            'colsample_bytree':[i/10 for i in range(5,11)],
            'colsample_bylevel':[i/10 for i in range(5,11)]
            }
xgb3=XGBRegressor(learning_rate=0.1,n_estimators=25,
                  min_child_weight=0.5,gamma=2,max_depth=2)
random_search=RandomizedSearchCV(xgb3,param_distributions=xgb_params,cv=10,
                                 n_iter=20,scoring='neg_mean_absolute_error',
                                n_jobs=-1,verbose=False)
random_search.fit(x_train,y_train)
report(random_search.cv_results_,3)
```

> Model with rank: 1
> Mean validation score: -1539.31339 (std: 416.70732)
> Parameters: {'subsample': 0.6, 'colsample_bytree': 0.6, 'colsample_bylevel': 0.6}

> Model with rank: 2
> Mean validation score: -1547.86783 (std: 420.01529)
> Parameters: {'subsample': 1.0, 'colsample_bytree': 0.7, 'colsample_bylevel': 0.8}

> Model with rank: 3
> Mean validation score: -1553.33132 (std: 411.66023)
> Parameters: {'subsample': 0.8, 'colsample_bytree': 0.8, 'colsample_bylevel': 0.6}

```
xgb_params={
            'reg_lambda':[i/10 for i in range(0,50)],
            'reg_alpha':[i/10 for i in range(0,50)]
            }
xgb4=XGBRegressor(n_estimators=25,min_child_weight=0.5,
                  gamma=2,max_depth=2, colsample_bylevel= 0.6,
                  colsample_bytree= 0.6, subsample= 0.6)
random_search=RandomizedSearchCV(xgb4,param_distributions=xgb_params,cv=10,
                                 n_iter=20,scoring='neg_mean_absolute_error',
                                n_jobs=-1,verbose=False)
random_search.fit(x_train,y_train)
report(random_search.cv_results_,3)
```

> Model with rank: 1
> Mean validation score: -1512.15494 (std: 450.56904)
> Parameters: {'reg_lambda': 3.5, 'reg_alpha': 2.3}

> Model with rank: 2
> Mean validation score: -1514.04499 (std: 434.73313)
> Parameters: {'reg_lambda': 2.2, 'reg_alpha': 4.1}

> Model with rank: 3
> Mean validation score: -1514.41400 (std: 435.72180)
> Parameters: {'reg_lambda': 2.3, 'reg_alpha': 3.0}

Considering the best parameter values thus obtained , our final model is :

```
xgb5=XGBRegressor(n_estimators=25,min_child_weight=0.5,
                  gamma=2,max_depth=2, colsample_bylevel= 0.6,
                  colsample_bytree= 0.6, subsample= 0.6,
                 reg_lambda=3.5,reg_alpha=2.3)

```

Lets check its performance using cross validation

```
from sklearn.model_selection import cross_val_score
scores=-
cross_val_score(xgb5,x_train,y_train,scoring='neg_mean_absolute_error',verbose=Fals
e,n_jobs=-1,cv=10)

```

```
scores
```

```
array([1382.17801316, 1141.28446396, 1065.85468616, 1041.07993432,
 888.63474221, 1595.63349977, 1977.38205667, 1851.14672852,
2266.21834198, 1913.92170945])
```

```
np.mean(scores)
```

```
1512.3334176203005
```

```
np.std(scores)
```

```
450.85032233797364
```

Couple comments for this specific problem here

- we should not build a complex model such as `Xgboost` for this small data
- Although the average performance is better than other alternatives , but the variation across data is huge
- performance is not very reliable, you should not be looking at just the average performance

Classification with Xgboost

We'll use the same sequential tuning of the parameters for the classification problem that we earlier solved with GBM

```
x_train=cd.drop('Y',1)
y_train=cd['Y']

from xgboost.sklearn import XGBClassifier
```

```
xgb_params = {
                "n_estimators":[100,500,700,900,1000,1200,1500]
            }
xgb1=XGBClassifier(subsample=0.8,colsample_bylevel=0.8,colsample_bytree=0.8)
grid_search=GridSearchCV(xgb1,cv=5,param_grid=xgb_params,
                         scoring='roc_auc',verbose=False,n_jobs=-1)
grid_search.fit(x_train,y_train)
report(grid_search.cv_results_,3)
```

```
Model with rank: 1
Mean validation score: 0.92749 (std: 0.00197)
Parameters: {'n_estimators': 500}
```

```
Model with rank: 2
Mean validation score: 0.92722 (std: 0.00170)
Parameters: {'n_estimators': 700}
```

```
Model with rank: 3
Mean validation score: 0.92683 (std: 0.00152)
Parameters: {'n_estimators': 900}
```

```
xgb_params={
            "gamma":[0,2,5,8,10],
            "max_depth": [2,3,4,5,6,7,8],
            "min_child_weight":[0.5,1,2,5,10]
            }
xgb2=XGBClassifier(n_estimators=500,subsample=0.8,
                   colsample_bylevel=0.8,colsample_bytree=0.8)
random_search=RandomizedSearchCV(xgb2,param_distributions=xgb_params,n_iter=20,
                                 cv=5,scoring='roc_auc',
                                 n_jobs=-1,verbose=False)
random_search.fit(x_train,y_train)
report(random_search.cv_results_,3)
```

> Model with rank: 1
> Mean validation score: 0.92711 (std: 0.00186)
> Parameters: {'min_child_weight': 2, 'max_depth': 6, 'gamma': 5}

> Model with rank: 2
> Mean validation score: 0.92676 (std: 0.00215)
> Parameters: {'min_child_weight': 1, 'max_depth': 8, 'gamma': 5}

> Model with rank: 3
> Mean validation score: 0.92670 (std: 0.00238)
> Parameters: {'min_child_weight': 1, 'max_depth': 6, 'gamma': 10}

```
xgb_params={
            'max_delta_step':[0,1,3,6,10],
            'scale_pos_weight':[1,2,3,4]
            }
xgb3=XGBClassifier(n_estimators=500,min_child_weight=2,gamma=5,max_depth=6,
                   subsample=0.8,colsample_bylevel=0.8,colsample_bytree=0.8)

grid_search=GridSearchCV(xgb3,param_grid=xgb_params,
                         cv=5,scoring='roc_auc',n_jobs=-1,verbose=False)

grid_search.fit(x_train,y_train)
report(grid_search.cv_results_,3)
```

> Model with rank: 1
> Mean validation score: 0.92796 (std: 0.00219)
> Parameters: {'max_delta_step': 0, 'scale_pos_weight': 2}

> Model with rank: 2
> Mean validation score: 0.92794 (std: 0.00219)
> Parameters: {'max_delta_step': 1, 'scale_pos_weight': 2}

> Model with rank: 3
> Mean validation score: 0.92787 (std: 0.00219)
> Parameters: {'max_delta_step': 3, 'scale_pos_weight': 2}

```
xgb_params={
            'subsample':[i/10 for i in range(5,11)],
            'colsample_bytree':[i/10 for i in range(5,11)],
            'colsample_bylevel':[i/10 for i in range(5,11)]
            }
xgb4=XGBClassifier(n_estimators=500,min_child_weight=2,gamma=5,max_depth=6,
                        scale_pos_weight=2,max_delta_step=0
                  )
random_search=RandomizedSearchCV(xgb4,param_distributions=xgb_params,
                                 cv=5,n_iter=20,scoring='roc_auc',
                                n_jobs=-1,verbose=False)
random_search.fit(x_train,y_train)
report(random_search.cv_results_,3)
```

> Model with rank: 1
> Mean validation score: 0.92858 (std: 0.00187)
> Parameters: {'subsample': 0.9, 'colsample_bytree': 0.5, 'colsample_bylevel': 1.0}

> Model with rank: 2
> Mean validation score: 0.92853 (std: 0.00218)
> Parameters: {'subsample': 0.8, 'colsample_bytree': 0.5, 'colsample_bylevel': 0.9}

> Model with rank: 3
> Mean validation score: 0.92838 (std: 0.00202)
> Parameters: {'subsample': 0.9, 'colsample_bytree': 0.9, 'colsample_bylevel': 1.0}

```
xgb_params={
            'reg_lambda':[i/10 for i in range(0,50)],
            'reg_alpha':[i/10 for i in range(0,50)]
            }
xgb5=XGBClassifier(n_estimators=500,min_child_weight=2,gamma=5,max_depth=6,
                        scale_pos_weight=2,max_delta_step=0,
                   colsample_bylevel=1.0, colsample_bytree= 0.5, subsample= 0.9)
random_search=RandomizedSearchCV(xgb5,param_distributions=xgb_params,
                                 cv=5,n_iter=20,scoring='roc_auc',
                                n_jobs=-1,verbose=False)

random_search.fit(x_train,y_train)
report(random_search.cv_results_,3)
```

> Model with rank: 1
> Mean validation score: 0.92874 (std: 0.00192)
> Parameters: {'reg_lambda': 0.7, 'reg_alpha': 0.3}

> Model with rank: 2
> Mean validation score: 0.92860 (std: 0.00195)
> Parameters: {'reg_lambda': 1.8, 'reg_alpha': 0.0}

> Model with rank: 3
> Mean validation score: 0.92853 (std: 0.00192)
> Parameters: {'reg_lambda': 0.9, 'reg_alpha': 0.8}

```
xgb6=XGBClassifier(n_estimators=500,min_child_weight=2,gamma=5,max_depth=6,
                    scale_pos_weight=2,max_delta_step=0,
                    colsample_bylevel=1.0, colsample_bytree= 0.5, subsample= 0.9,
                   reg_lambda=0.7,reg_alpha=0.3)
scores=cross_val_score(xgb6,x_train,y_train,scoring='roc_auc',
                        verbose=False,n_jobs=-1,cv=10)

```

```
np.mean(scores)
```

> 0.9292637432983983

```
np.std(scores)
```

> 0.0030315969143107297

We'll conclude our discussion here on boosting machines . For making predictions on new data, same functions on the model object `predict` and `predict_proba` can be used . As usual you'll need to ensure that test data on which you want to make predictions has same columns and type as in the training data which was passed to the algorithm while training.

Note : different runs might lead to slightly different parameter values , however the performance wouldn't be wildly different

Chapter 7 : KNN, Naive Bayes and SVM

We will cover the following points in this discussion:

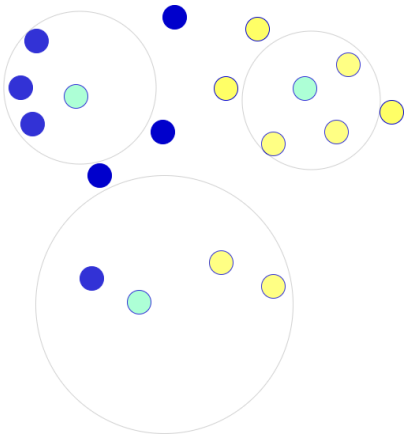
- K-Nearest Neighbors (KNN)
- Naive Bayes
- Support Vector Machines (SVM)
- Implementation in Python using Scikit-learn library - specifically for text data

K-Nearest Neighbours (KNN)

KNN is one of the simplest supervised learning algorithms and it can be used for both, classification as well as regression.

In KNN for classification, an observation is classified by a majority vote of its neighbors. This observation is assigned to a class most common amongst its k nearest neighbors. k is a small positive number which stands for the number of nearest observations from which we wish to take a vote.

For regression, the difference is that instead of taking a vote from the k nearest neighbors, the algorithm considers averages of the k nearest neighbors.



Let us understand this better with an example.

In the diagram above, our objective is to classify the light blue circles i.e. determine whether they should be dark blue or yellow considering its closest neighbors. We choose k=3 i.e we will consider the 3 closest neighbors to determine which class the light blue circle belongs to. In the diagram, you will observe a bigger circle drawn around each of the light blue points which include the three circles closest to it.

Lets consider the top left bigger circle. For the light blue circle within it, we observe that its three closest neighbors are dark blue. Hence, we will consider the majority vote and classify the light blue circle as dark blue.

Now, considering the top right bigger circle, the three closest neighbors for the light blue circle are yellow, hence this light blue circle will be classified as a yellow circle.

However, observing the bottom bigger circle, the light blue circle has two yellow circles and one dark blue circle as its closest neighbors. Since yellow circles are in a majority here, it will be classified as a yellow circle.

Having said that, for the bottom circle, despite the majority being the yellow circle, we observe that the dark blue circle is closer to the light blue circle. What if we want to consider the distance of the nearest neighbors to the light blue circle to determine it class also? To take care of this, we can consider the inverse of the distance as weights to the votes. In this case, the dark blue circle will be given a higher weight, say 0.7 since it is closer to the light blue circle, and the two yellow balls will be given lesser weight, say 0.3 and 0.25, since they are further away from the light blue circle. Now

when we consider the weights for the two classes, the dark blue circles will have a weight of 0.7 and the yellow circles will have a weight of 0.55 (0.3+0.25). Considering this, the dark blue class will have a higher vote now as compared to the two yellow circles combined and hence the light blue circle will be classified as dark blue one. In short, if we consider a weighted majority vote instead of a simple majority vote, our classification may be different.

One thing to note here is that if we change the value of 'k' i.e. the number of closest neighbors to consider, then the classification of the light blue balls may change depending on the neighborhood size.

The steps followed in the algorithm are as follows:

- Read the data
- Set the value of k
- To predict the class of an observation from the test dataset, need to iterate through all the training data points
- Calculate the distance between test observation and each training data point. Distance measures that can be used are Euclidean distance, Chebyshev, cosine similarity etc.
- Calculated distances are sorted in ascending order
- Consider only the top k rows from the sorted array
- Get the most frequent class from these k rows
- The most frequent class is the predicted class

Implementation of K-Nearest Neighbours using scikit-learn

```
import numpy as np
import pandas as pd
```

```
# load the iris dataset
from sklearn.datasets import load_iris
iris = load_iris()
```

```
# store the feature matrix (X) and response vector (y)
X = iris.data
y = iris.target
```

```
# splitting X and y into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=2)
```

Since KNN relies on a distance measure to figure out which class a test observation belongs to. Scaling the data becomes a must here since if we do not scale the data then features with larger values will be dominant.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)

X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

We import the KNeighborsClassifier to classify the data, using 'k' as 5 i.e. the algorithm will consider 5 nearest neighbors to assess class membership.

```
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors=5)
classifier.fit(X_train, y_train)
```

```
y_pred = classifier.predict(X_test)
```

```
# comparing actual response values (y_test) with predicted response values (y_pred)
from sklearn import metrics
print("KNN model accuracy:", metrics.accuracy_score(y_test, y_pred))
```

> KNN model accuracy: 0.9666666666666667

Reference: https://stackabuse.com/k-nearest-neighbors-algorithm-in-python-and-scikit-learn/

Points to note:

- The KNN algorithm is purely distance based and variables which have larger values i.e. are high on scale will dominate. In order to avoid this, we will need to standardize the variables such that all the variables are on a single scale.
- Votes can be weighted in different ways, inverse of the the distance is most popular.
- Lower values of k will capture very niche patterns and will tend to over-fit.
- Very high values of k will capture broad patterns but may miss local patterns in the data.
- There is no model equation here; the prediction depends completely on the training data. e.g. when we take a new observation to be classified, we determine its class by checking the classes of the training data observations present in its neighborhood. Hence, training data itself is the model.
- KNN can capture very local patterns in the data whereas other algorithms can capture patterns across the data. Standalone KNN is not a very good algorithm because it relies on simple neighborhood voting. There is no general pattern extraction from different variables. So they are rarely used on a standalone basis. But since they can extract very niche patterns, it is very useful in stacking algorithms.

Naive Bayes

Naive Bayes is a classification algorithm based on Bayes Theorem with a naive assumption of independence among features i.e. presence of one feature does not affect the other. These models are easy to build and very useful for high dimensional datasets.

The fundamental Naive Bayes assumption is that each feature makes an independent and equal contribution to the outcome. This assumption is not generally correct in the real world but works quite well in practice.

Lets quickly review some concepts of probability.

Say we have a bag with 10 balls, 6 blue and 4 red. If we choose a ball from this bag, then the probability that it is blue will be 6/10 i.e. the number of blue balls divided by all the possible selections and similarly the probability that it will be red will be 4/10.

Lets say, we have a pack of cards of 52 cards. If we choose a card from this pack at random, then, say for event A the probability that it is a 5 is 4/52 i.e. the number of 5's present in the pack divided by the total number of cards. And, say for event B, the probability that it is red is 26/52 i.e. total number of reds in the pack of cards divided by the total number of cards.

$$P(A) = \frac{|A|}{|U|} \; and \; P(B) = \frac{|B|}{|U|}$$
$$where \; |A| \; = \; number \; of \; times \; 5 \; appears \; in \; the \; pack \; of \; cards \; = \; 4$$
$$and \; |B| \; = \; number \; of \; red \; cards \; in \; the \; pack \; of \; cards \; = \; 26$$
$$and \; |U| \; = \; total \; numer \; of \; cards \; = \; 52$$

Now, instead of seeing two events in isolation, lets look at them together i.e lets consider the probability of selecting a 5 which is red. In other words, we wish to find the probability when both the events A and B happen together.

$$P(AB) = \frac{|AB|}{|U|}$$

We know that in the pack of cards, we have two 5's that are red as well. So the probability of selecting a 5 which is red as well is 2/52.

Now, let say we want to find the probability of selecting a 5 given that we already have the red cards P(A|B) i.e the probability of an event A happening given that event B has happened already. All possible outcomes in this case will be the number of red cards i.e. count of B - 26 red cards. Within this subset we select a 5 that is red i.e. both the events A and B have occurred. Therefore the probability that 5 is selected given that we already have the red cards only is 2/26.

$$P(A|B) = \frac{|AB|}{|B|}$$

$$Dividing\ the\ numerator\ and\ denominator\ by\ total\ number\ of\ cards$$

$$P(A|B) = \frac{|AB|/|U|}{|B|/|U|}$$

$$Referring\ to\ the\ equations\ above$$

$$P(A|B) = \frac{P(AB)}{P(B)} => P(AB) = P(A|B) * P(B)$$

$$Similarly\ the\ probability\ of\ event\ B\ happening\ given\ event\ A\ has\ happened\ already$$

$$P(B|A) = \frac{P(AB)}{P(A)} => P(AB) = P(B|A) * P(A)$$

$$Equating\ the\ two\ equations$$

$$P(A|B) * P(B) = P(B|A) * P(A)$$

We now get the Bayes theorem:

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$$

Now, instead of a single factor B, what if we have multiple factors i.e. B1, B2, B3, etc. We want to know what is the probability of getting an A given multiple factors B1, B2 etc. To handle this we can extend the Bayes Theorem as follows:

$$P(A|B_1 \cap B_2 \cap B_3 \dots) = \frac{P(B_1 \cap B_2 \cap B_3 \dots |A) * P(A)}{P(B_1 \cap B_2 \cap B_3 \dots)}$$

This basically means that given certain factors B1, B2, B3 etc, what is the probability of getting the outcome A.

Now, when we consider Naive Bayes, we say that each of these factors B1, B2, B3 etc given that A has happened already, are independent of each other, hence we can write the above equation as follows:

$$P(A|B_1 \cap B_2 \cap B_3 \dots) = \frac{P(B_1|A) * P(B_2|A) * P(B_3|A) \dots * P(A)}{P(B_1 \cap B_2 \cap B_3 \dots)}$$

Note: The denominator will stay constant, it will not change with A.

For a detailed example with numbers, please visit the following link: https://stackoverflow.com/questions/10059594/a-simple-explanation-of-naive-bayes-classification or refer to the class presentation.

Types of Naive Bayes Classifiers:

- Gaussian Naive Bayes: In Gaussian Naive Bayes, continuous values associated with each feature are assumed to be distributed according to a Gaussian distribution.
- Multinomial Naive Bayes: Features have discrete values. This is primarily used for document classification. In this case the features used are the frequency of the words present in the document.

- Bernoulli Naive Bayes: This is similar to the multinomial naive Bayes but the predictors are Boolean variables. This is also popular for document classification tasks.

Implementation of Gaussian Naive Bayes Classifier using scikit-learn

To implement Naive Bayes, we will use the same iris dataset we used earlier.

```
# load the iris dataset
from sklearn.datasets import load_iris
iris = load_iris()
```

```
# store the feature matrix (X) and response vector (y)
X = iris.data
y = iris.target
```

```
# splitting X and y into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=1)
```

```
# training the model on training set
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
gnb.fit(X_train, y_train)
```

```
GaussianNB(priors=None, var_smoothing=1e-09)
```

```
# making predictions on the testing set
y_pred = gnb.predict(X_test)
```

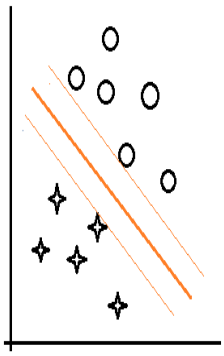
```
# comparing actual response values (y_test) with predicted response values (y_pred)
from sklearn import metrics
print("Gaussian Naive Bayes model accuracy:", metrics.accuracy_score(y_test,
y_pred))
```

> Gaussian Naive Bayes model accuracy: 0.9666666666666667

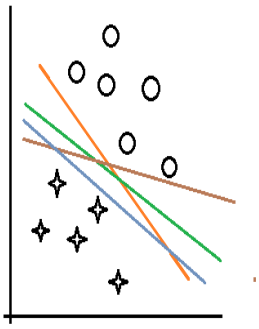
Reference: https://www.geeksforgeeks.org/naive-bayes-classifiers/

Support Vector Machines (SVM)

SVM is one of the supervised learning algorithms using training data to learn a model and make predictions. It learns a linear model. In two dimensions, the algorithm outputs a line which categorizes the data whereas in higher dimensions it generates a hyper-plane to categorize the data.

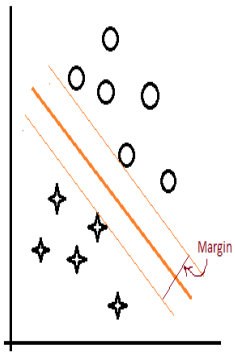


Considering the diagram above, we observe that the orange line classifies the two classes i.e. stars and circles. Any observation that lies to the left of the line falls in the star class and any observation to the right of the line falls under the circle class. In short, SVM separates the classes using a line or a hyper-plane (for higher dimensions).

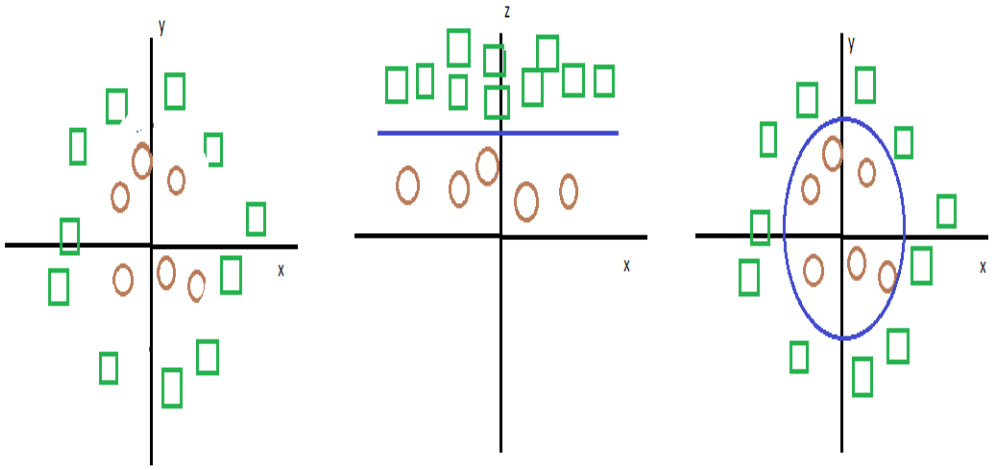


Many hyper-planes can be considered to categorize the two classes completely. However, using SVM, our objective is to find the hyper-plane which has maximum distance from points of either class. Maximizing the distance increases the confidence that the future data points will be categorized correctly as well.

This distance between the hyper-plane and the nearest data points is called 'margin'. Margin helps the algorithm decide the optimal hyper-plane.



Now, lets consider what happens when the classes cannot be separated by a line/hyper-plane.



No linear boundary separates the two classes

Transformation applied and one more dimension 'z' added; data now linearly separable

Transformed back to the xy plane; linear separator becomes a circle

Considering the leftmost diagram above, we can see that a linear boundary cannot categorize the two classes in the x-y plane. In order to be able to categorize the classes, another dimension z is added as can be observed from the middle figure. We can see that a clear separation is visible and a line can be used to categorize the two classes. When we now transform the data back into the x-y plane, the linear boundary becomes a circle as can be observed from the right most figure. These transformations are called 'kernels'. Using this 'kernel trick', the algorithm takes a low dimensional input and converts it into a higher dimensional space; thereby converting a non-separable problem to a separable one.

In order to understand the math behind SVM, you may want to refer to the following excellent link: https://www.svm-tutorial.com/2014/11/svm-understanding-math-part-1/

Next we will discuss about parameter used in SVM. There are three parameters we primarily tune in this algorithm:

- kernel - It defines the a distance measure between new data and the support vectors i.e. observations closest to the hyper-plane. The dot product is the similarity measure used for a linear kernel since the distance is a linear combination of the inputs. When we consider higher dimensions other kernels such as a Polynomial Kernel and a Radial Kernel can be used that transform the input space into higher dimensions.
- C (Regularization parameter) - When the value of C is large, smaller-margin hyper-plane will be

considered since it stresses on getting all the training points classified correctly. On the other hand, a small value of C will consider a larger margin hyper-plane, even if some points are misclassified by the hyper-plane.

- gamma - The gamma parameter defines how far the influence of a each training observation affects the calculation of the optimal hyper-plane. Low gamma values consider points even if they far away from the plausible hyper-plane and high gamma values consider the points which are closer to the plausible hyper-plane to get the optimal hyper-plane.

```
# load the iris dataset
from sklearn.datasets import load_iris
iris = load_iris()
```

```
# store the feature matrix (X) and response vector (y)
X = iris.data
y = iris.target
```

```
# splitting X and y into training and testing sets
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=1)
```

```
# training the model on training set
from sklearn import svm
# we create an instance of SVM
C = 1.0 # SVM regularization parameter
svc = svm.SVC(kernel='linear', C=1, gamma='auto')
```

```
svc.fit(X_train, y_train)
```

```
# making predictions on the testing set
y_pred = svc.predict(X_test)
```

```
# comparing actual response values (y_test) with predicted response values (y_pred)
from sklearn import metrics
print("SVM model accuracy:", metrics.accuracy_score(y_test, y_pred))
```

> SVM model accuracy: 1.0

KNN, Naive Bayes and SVM implementation in Python for text data

The data we consider now is different than the structured data we have used till now. The data are multiple text files and each text file contains an article. The articles belong to different categories like crude, money and foreign exchange etc. What do we do if we are given a new article and we need to classify it into any of these categories? There are two issues to be handled here: first, we do not have our data in a single file. We need to bring data into a single file which can be used for further analysis. Secondly, the only data we have here is the text in the articles. How do we convert the text into a format which can be used by the machine learning algorithms? We need to convert the data into features such that we can make use of the algorithms for classification.

Lets start with reading in the data.

```
import os
path=r"\Users\anjal\Dropbox\PDS V3\Data\reuters_data"
```

We will start with collecting all the data in a single file first.

```
files=os.listdir(path)
```

The os.listdir(path) function returns all the files present in the folder specified by the 'path' argument.

We can observe the file names as follows:

```
files[0:10]
```

```
['.ipynb_checkpoints',
'.RData',
'.Rhistory',
'training_crude_10011.txt',
'training_crude_10078.txt',
'training_crude_10080.txt',
'training_crude_10106.txt',
'training_crude_10168.txt',
'training_crude_10190.txt',
'training_crude_10192.txt']
```

We want content only from the .txt files; however, we can see that some other files are present here as well. We can clean them up i.e. keep only those files which have a .txt in their file name.

```
files=[x for x in files if '.txt' in x]
```

Now if we look at files, the unnecessary files are not there anymore.

```
files[0:10]
```

```
['training_crude_10011.txt',
'training_crude_10078.txt',
'training_crude_10080.txt',
'training_crude_10106.txt',
'training_crude_10168.txt',
'training_crude_10190.txt',
'training_crude_10192.txt',
'training_crude_10200.txt',
'training_crude_10228.txt',
'training_crude_1026.txt']
```

We clean the text for all the files as follows:

```
target=[]
article_text=[]
for file in files:
    if '.txt' not in file:continue # do not read the content from a file not having .txt in its name
    f=open(path+'\\'+file,encoding='latin-1') # for every file a handle is created
    article_text.append(" ".join([line.strip() for line in f if line.strip()!=""])) # removes lines without text using strip()

 # and returns a single string for each article
    if "crude" in file: # if the file name has crude, then target list is appended with 'crude' else with 'money'
        target.append("crude")
    else:
        target.append("money")
    f.close()
```

Now we bind the two lists created above into a dataframe.

```
mydata=pd.DataFrame({'target':target,'article_text':article_text})
```

The dataframe 'mydata' consists of two columns, the 'article_text' column containing the text and the 'target' column consisting of the topic of the text.

```
mydata.head()
```

	target	article_text
0	crude	CANADA OIL EXPORTS RISE 20 PCT IN 1986 Canadia...
1	crude	BP <BP> DOES NOT PLAN TO HIKE STANDARD <...
2	crude	BP<BP> OFFER RAISES EXPECTATIONS FOR OIL VA...
3	crude	USX <X> SAYS TALKS ENDED WITH BRITISH PETRO...
4	crude	BP <BP> MAY HAVE TO RAISE BID - ANALYSTS B...

```
mydata.shape
```

```
(927, 2)
```

The data has 927 rows and 2 columns.

We can access any of the articles as follows:

```
mydata['article_text'][0]
```

```
'CANADA OIL EXPORTS RISE 20 PCT IN 1986 Canadian oil exports rose 20 pct in 1986 over the previous year to 33.96 mln cubic meters, while oil imports soared 25.2 pct to 20.58 mln cubic meters, Statistics Canada said. Production, meanwhile, was unchanged from the previous year at 91.09 mln cubic feet. Natural gas exports plunged 19.4 pct to 21.09 billion cubic meters, while Canadian sales slipped 4.1 pct to 48.09 billion cubic meters. The federal agency said that in December oil production fell 4.0 pct to 7.73 mln cubic meters, while exports rose 5.2 pct to 2.84 mln cubic meters and imports rose 12.3 pct to 2.1 mln cubic meters. Natural gas exports fell 16.3 pct in the month 2.51 billion cubic meters and Canadian sales eased 10.2 pct to 5.25 billion cubic meters.'
```

```
mydata['target'].value_counts()
```

```
money    538
crude    389
Name: target, dtype: int64
```

We see that out of the 927 articles, 538 belong to the 'money' category and 389 belong to the 'crude' category.

To find the tentative performance on our model we will break the dataset into training and validation parts.

```
from sklearn.model_selection import train_test_split
article_train,article_test= train_test_split(mydata,test_size=0.2,random_state=2)
article_train.head()
```

	target	article_text
280	crude	MALAYSIA TO CUT OIL OUTPUT FURTHER, TRADERS SA...
688	money	BANKERS OPPOSE STRICT TAIWAN CURRENCY CONTROLS...
375	crude	OPEC WITHIN OUTPUT CEILING, SUBROTO SAYS Opec ...
665	money	U.K. MONEY MARKET SHORTAGE FORECAST AT 300 MLN...
589	money	CURRENCY FUTURES TO KEY OFF G-5, G-7 MEETINGS ...

```
article_train.reset_index(drop=True,inplace=True)
```

```
article_train.head()
```

	target	article_text
0	crude	MALAYSIA TO CUT OIL OUTPUT FURTHER, TRADERS SA...
1	money	BANKERS OPPOSE STRICT TAIWAN CURRENCY CONTROLS...
2	crude	OPEC WITHIN OUTPUT CEILING, SUBROTO SAYS Opec ...
3	money	U.K. MONEY MARKET SHORTAGE FORECAST AT 300 MLN...
4	money	CURRENCY FUTURES TO KEY OFF G-5, G-7 MEETINGS ...

```
y_train=(article_train['target']=='money').astype(int)
y_test=(article_test['target']=='money').astype(int)
```

Now we have the data in a column format in a single file. However, we have still not created features that can be used by the different machine learning algorithms for classification of articles.

One simple idea is that we consider every word across all the articles i.e. create a dictionary containing all the distinct words present across all the articles. We can then consider a word from the dictionary and count the number of times it is present in each article; e.g. considering the word 'currency', we can count the number of times it is present in each article and store this count. The count of different words can be used as features i.e. count features.

When doing this, we will come across many words that do not contribute to the article belonging to one category or the other. These words are called as stopwords e.g the, to, a etc. Such words are not useful for differentiating the articles and can be removed.

Also, we may want all words to be converted to their base form i.e. both 'playing' and 'played' should be converted to 'play'. This is referred to as lemmatization. We will use the WordNetLemmatizer for this.

We also want to remove punctuation from the text else punctuation will be considered as separate features.

```
from nltk.stem.wordnet import WordNetLemmatizer
from nltk.corpus import stopwords
from string import punctuation
from nltk.tokenize import word_tokenize # used to break sentences into words
lemma = WordNetLemmatizer()
my_stop=set(stopwords.words('english')+list(punctuation)) # we may want to
add/remove words depending on the business context
```

```
# Function to clean the text data
def split_into_lemmas(message):
    message=message.lower() # converts all text to lowercase
    words = word_tokenize(message)
    words_sans_stop=[]
    for word in words :
        if word in my_stop:continue
        words_sans_stop.append(word) # gets all words from the message except the stopwords
    return [lemma.lemmatize(word) for word in words_sans_stop] # lemmatized words returned
```

```
from sklearn.feature_extraction.text import CountVectorizer
```

CountVectorizer tokenizes the text and builds a vocabulary of words which are present in the text body.

The CountVectorizer function below uses the following parameters:

- 'analyzer = split_into_lemmas' sends the text to the function split_into_lemmas() and then gets every word which is cleaned.
- 'min_df = 20' argument is used so that only those words are considered which are present in the data at least 20 times.
- 'max_df = 500' indicates that the words which are present more than 500 times in the data will not be considered.
- 'stop_words = my_stop' argument takes the stop words defined earlier as its input. CountVectorizer just counts the occurrences of each word in its vocabulary. Hence common words like ???the???, ???and???, 'a' etc. become very important features since their frequency is high even though they add little meaning to the text. The words considered as stop words are not used as features.

```
tf=
CountVectorizer(analyzer=split_into_lemmas,min_df=20,max_df=500,stop_words=my_stop)
```

The fit() function is used to learn the vocabulary from the training text and the transform() function is used to encode each article text as a vector. This encoded vector has a length of the whole vocabulary and an integer count for the number of times each word appeared in the article text is returned for each article in this vector.

```
tf.fit(article_train['article_text'])
```

```
train_tf=tf.transform(article_train['article_text'])
```

```
train_tf
```

```
<741x677 sparse matrix of type '<class 'numpy.int64'>'
    with 36338 stored elements in Compressed Sparse Row format>
```

```
print(train_tf.shape)
print(type(train_tf))
print(train_tf.toarray())

```

```
(741, 677)
<class 'scipy.sparse.csr.csr_matrix'>
[[0 5 0 ... 0 0 0]
 [0 0 0 ... 0 0 0]
 [2 2 0 ... 0 0 0]
 ...
 [0 1 0 ... 0 0 0]
 [0 1 0 ... 0 0 0]
 [1 6 0 ... 0 1 0]]
```

We can then see that the encoded vector is a sparse matrix.

We observe that the array version of the encoded vector shows counts for different words.

```
x_train_tf = pd.DataFrame(train_tf.toarray(), columns=tf.get_feature_names())
print(x_train_tf.head())
```

```
   ''  's  --  ...  1  1.5  10  100  15  15.8  ...   work  working  world  \
0   0   5   0    0  1    0   2    0   0     0  ...      0        0      0
1   0   0   0    0  0    0   0    0   0     0  ...      0        0      0
2   2   2   0    0  0    0   0    0   0     2  ...      0        0      0
3   0   0   0    0  0    0   0    1   0     0  ...      0        0      0
4   7   1   2    0  0    0   0    0   0     0  ...      0        0      0

   worth  would  year  yen  yesterday  yet  york
0      0      3     1    0          0    0     0
1      0      1     0    0          0    0     0
2      0      3     0    0          0    0     0
3      0      3     0    0          0    0     0
4      0      5     0    7          0    0     0

[5 rows x 677 columns]
```

```
test_tf=tf.transform(article_test['article_text'])
```

```
test_tf.toarray()
```

```
array([[2, 1, 0, ..., 0, 1, 0],
       [0, 1, 0, ..., 0, 0, 0],
       [2, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 3, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [3, 1, 0, ..., 0, 0, 0]], dtype=int64)
```

```
x_test_tf=pd.DataFrame(test_tf.toarray(), columns=tf.get_feature_names())
```

```
x_test_tf.head()
# display in pdf will truncated on the right hand side
```

	''	's	--	...	1	1.5	10	100	15	15.8	...	work	working	world	worth	would	yea
0	2	1	0	0	0	0	1	1	0	0	...	0	0	0	0	2	0
1	0	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0
2	2	0	0	0	0	0	0	0	0	0	...	0	0	1	0	1	0
3	0	3	0	0	0	0	0	0	0	0	...	0	0	0	0	0	7
4	0	1	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0

5 rows ?? 677 columns

In the code below you will notice that the number of columns created are 677 which is quite a lot. Text based features usually end up being too many.

```
print(x_train_tf.shape)
print(x_test_tf.shape)
```

```
(741, 677)
(186, 677)
```

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.svm import SVC
from sklearn.naive_bayes import MultinomialNB
```

KNN

```
knn=KNeighborsClassifier(n_neighbors=10)
```

```
knn.fit(x_train_tf,y_train)
```

```
predictions=knn.predict(x_test_tf)
```

```
accuracy_score(y_test,predictions)
```

```
0.9623655913978495
```

SVM

```
clf_svm=SVC()
```

```
clf_svm.fit(x_train_tf,y_train)
```

```
accuracy_score(y_test,clf_svm.predict(x_test_tf))
```

```
0.978494623655914
```

Naive Bayes

```
clf_nb=MultinomialNB()
```

```
clf_nb.fit(x_train_tf,y_train)
```

```
accuracy_score(y_test,clf_nb.predict(x_test_tf))
```

```
0.9892473118279570
```

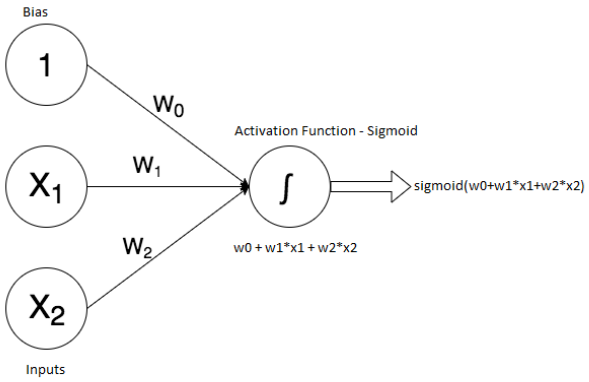
We observe that Naive Bayes performs better than the rest of the algorithms in text classification for this example. This does not mean that this will be the case for every problem. We should anyway use multiple algorithms and chose the one which performs best for our particular problem in discussion. Another point for this particular discussion, notice that we haven't tuned parameters. You can follow similar processes as used in earlier modules to do so.

Chapter 8 :Neural Networks

We will cover the following points in our discussion:

- Neural Network Basics
- Forward and Backward propagation with a numerical example
- Neural Network Implementation in Python using scikit learn - MLPClassifier

We will start with discussing how the neural networks are represented.



We can see that there are three things that make up a neural network:

- the neurons or nodes or units
- the connections or weights or parameters
- the bias

These are the basic building blocks of the neural network.

Referring to the diagram above, we can see that there are two input variables: x1 and x2 and a bias term with a constant value 1. Each of these inputs are multiplied by the weights w0, w1 and w2. It then sums the multiplication and passes the sum $w0 + w1 * x1 + w2 * x2$ to an activation function - introducing non-linearity in the model. In the example above the non-linearity is introduced using the sigmoid function. This process results in a single output from a neuron i.e. $\sigma(w0 + w1 * x1 + w2 * x2)$.

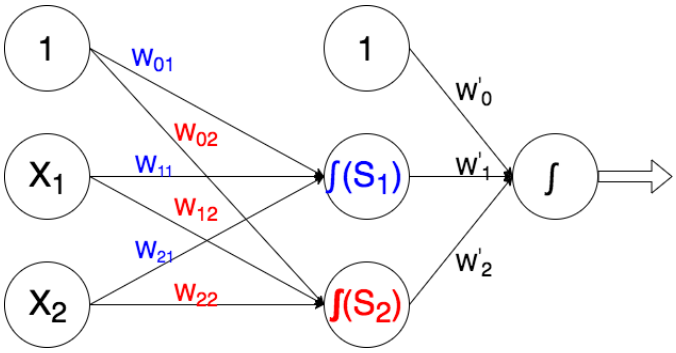
In the figure above, there are two layers: the input layer and the output layer.

Why do we need to introduce non-linearity in the network or why do we need to use an activation function?

1. To project values having an infinite range (-inf to +inf) i.e. linear function to probabilities where the value range may be from 0 to 1 or -1 to +1 or something else depending on the activation function used.
2. These functions introduce non-linearity which enables the detection of non-linear patterns in the data. If only linear activation functions are used in the network, a combination of these linear functions will also be a linear function.
3. Activation functions decide whether a neuron fires or not, provided its value crosses a certain threshold.

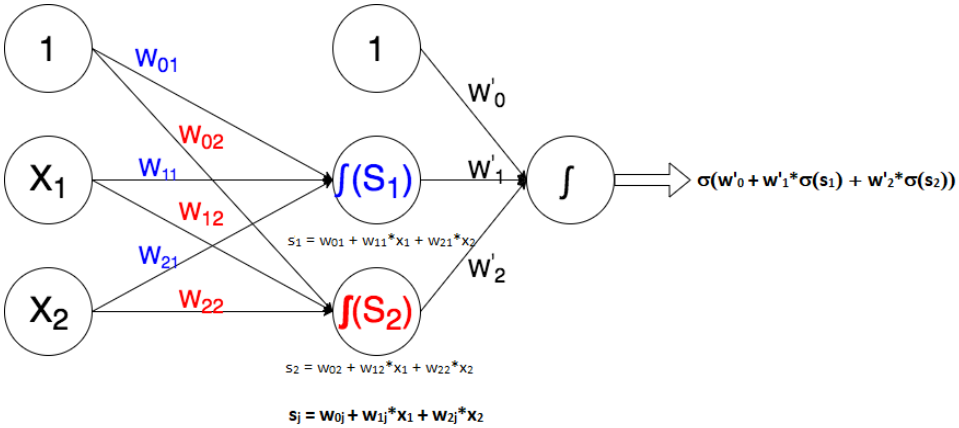
For details please refer to the following link: https://towardsdatascience.com/activation-functions-and-its-types-which-is-better-a9a5310cc8f

Usually there will be more layers present as shown in the figure below. The layers between the input and output layers are referred to as hidden layers. In the figure below, we have three layers: the input layer, a single hidden layer and an output layer. Note: hidden nodes are not connected directly to the input data or the eventual output. We can have any number of hidden layers as well as any number of nodes in each layer.



Bias node

We will now discuss about bias nodes. These nodes are added to neural networks to help the network learn patterns. They act as an input node and always contain a constant value; can be 1 or some other value. Due to this they are not connected to any previous layer. For understanding bias further, you may refer to the following post: https://www.quora.com/What-is-bias-in-artificial-neural-network



Hidden layer

Lets consider the hidden layer in the diagram above. The first node in the hidden layer is the bias node. The second node, takes as its input, weighted inputs from all the nodes in the previous layer i.e.$s_1 = w_{01} + w_{11} * x_1 + w_{21} * x_2$ and applies an activation on this linear combination of inputs. In this case, we are using the sigmoid activation function.

$h_1 = \sigma(w_{01} + w_{11} * x_1 + w_{21} * x_2)$.

Similarly, for the third node we get the following output:

$h_2 = \sigma(w_{02} + w_{12} * x_1 + w_{22} * x_2)$.

Here, s_1 and s_2 are what come as input in each node of the hidden layer and what goes out is h_1 and h_2 i.e. sigmoid function applied to s_1 and s_2.

Output layer

For the third layer i.e. the output layer, we get a weighted linear combination of all the nodes of the previous layer i.e. $\sigma(s_1)$ and $\sigma(s_2)$ along with the bias term.

$output = \sigma(w'_0 + w'_1 * \sigma(s_1) + w'_2 * \sigma(s_2))$

We need to keep in mind that as we keep adding layers the output function keeps getting more complex.

Neural network is a very powerful idea. It can model any non-linear pattern present in the data.

"A feed-forward network with a single layer is sufficient to represent any function, but the layer may be in-feasibly large and may fail to learn and generalize correctly." -??????Ian Goodfellow, DLB

Neural networks may not perform very well if the dataset is too small. But with sufficient data points neural networks function very well.

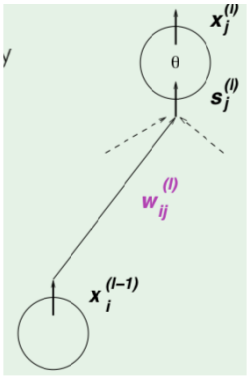
General representation of Neural Networks

Notation for neural networks is as shown below:

$$w_{ij}^{(l)} \quad \begin{cases} 1 \le l \le L & \text{layers} \\ 0 \le i \le d^{(l-1)} & \text{inputs} \\ 1 \le j \le d^{(l)} & \text{outputs} \end{cases}$$

$$x_j^{(l)} = \theta(s_j^{(l)}) = \theta\left(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)}\, x_i^{(l-1)}\right)$$

Reference: http://work.caltech.edu/slides/slides10.pdf



l — *layers*; *ranges from* 1 *to* L *i.e.* 1 — *input layer*, L — *output layer*, l — *hidden layers*

i — *inputs* : *this value goes from* 0 *to the number of nodes in that layer*

j — *outputs* : *this value goes from* 1 *to the number of nodes in that layer since constant not connected from the previous layer*

$d^{(l)}$ — *number of nodes in the* l^{th} *layer*

$x_i{}^{(l-1)}$ — *input from* i^{th} *node in the* $(l-1)^{th}$ *layer*

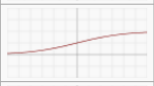
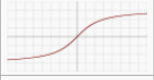
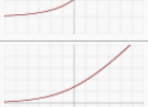
$w_{ij}{}^{(l)}$ — *weight of the connection from* $(l-1)^{th}$ *layer to the* l^{th} *layer connecting* i^{th} *to* j^{th} *node*

$s_j{}^{(l)}$ — *linear combination of weighted inputs that goes into the* j^{th} *node of* l^{th} *layer*

$x_j{}^{(l)}$ — *output obtained after activation/non − linearity applied to* $s_j{}^{(l)}$

Activation Functions

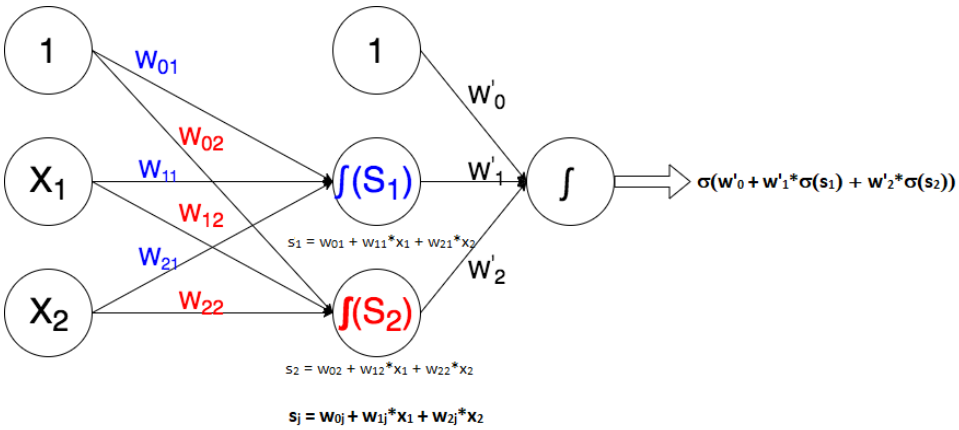
Till now we have considered the sigmoid activation function only. There are a number of other activation functions that can be used as well.

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \ge 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \ne 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \ge 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \ge 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU)[2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \ge 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \ge 0 \end{cases}$
Exponential Linear Unit (ELU)[3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \ge 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \ge 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1+e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

Reference: https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

How does a neural network work?

Lets try and understand how a neural network works with the help of an example.



The example uses a neural network with two inputs, one output and two hidden neurons. Our inputs are 0.05 and 0.10 and output is 0.01.

Consider the following random values given to each of the weights: $x_1 = 0.05???$, $x_2 = 0.10???$, $w_{01} = 0.35???$,
$w_{11} = 0.15???$, $w_{21} = 0.20???$, $w_{12} = 0.25???$, $w_{22} = 0.30???$, $w'_0 = 0.6???$, $w'_1 = 0.4???$,
$w'_2 = 0.45???$, $overall\ output = 0.01???$.
Overall output is the actual value against which we will compare our predicted value.

Forward Pass

Given the inputs, bias and the randomly initialized weights, we can check what prediction is done by the neural network moving from left to right.

Lets begin.

We will first compute the weighted linear combination of the inputs $x_1???$ and $x_2???$ from the first layer. i.e. $\mathbf{s_1 = w_{01} + w_{11} * x_1 + w_{21} * x_2} \cdots (1)???$

and

$\mathbf{s_2 = w_{02} + w_{12} * x_1 + w_{22} * x_2} \cdots (2)$

$s_1???$ and $s_2???$ are sent as an input to the next layer i.e. the hidden layer. Here an activation function is applied introducing non-linearity and squashing the input i.e. $h_1 = \sigma(s_1)???$ and $h_2 = \sigma(s_2)???$.

This process is then repeated for the output layer as well.

Starting with forward propagation from the input layer.

Calculating the linear combination of inputs i.e $s_1???$:

$$\mathbf{s_1} = \mathbf{w_{01} * 1 + w_{11} * x_1 + w_{21} * x_2} \cdots (3)$$
$$s_1 = 0.35 * 1 + 0.15 * 0.05 + 0.20 * 0.10 = 0.3775$$

Applying the activation function, in this case the sigmoid function to $s_1???$ gives $h_1???$ as follows:

$$\mathbf{h_1} = \sigma(\mathbf{s_1}) = \frac{\mathbf{1}}{\mathbf{1 + e^{-s_1}}} \cdots (4)$$
$$h_1 = \frac{1}{1 + e^{-0.3775}} = 0.593269992$$

Similarly, we apply the sigmoid function to $s_2???$ which gives $h_2???$ as follows:

$$\mathbf{h_2} = \sigma(\mathbf{s_2}) = \frac{\mathbf{1}}{\mathbf{1} + \mathbf{e}^{-\mathbf{s_2}}} \cdots (5)$$
$$h_2 = 0.596884378$$

Note: The activation function need not necessarily be sigmoid. It can be any of the activation functions discussed before.

We repeat this process for the output layer too.

$$\mathbf{o_{in}} = \mathbf{w_0'} * \mathbf{1} + \mathbf{w_1'} * \mathbf{h_1} + \mathbf{w_2'} * \mathbf{h_2} \cdots (6)$$
$$o_{in} = 0.6 * 1 + 0.4 * 0.593269992 + 0.45 * 0.596884378 = 1.105905967$$

Applying the sigmoid function to o_{in}??? we get o_{out}??? as follows:

$$\mathbf{o_{out}} = \sigma(\mathbf{o_{in}}) = \frac{\mathbf{1}}{\mathbf{1} + \mathbf{e}^{-\mathbf{o_{in}}}} \cdots (7)$$
$$o_{out} = \frac{1}{1 + e^{1.105905967}} = 0.75136507$$

After we get the prediction i.e. o_{out}??? made by the first pass of the neural network using random weights, we calculate the total error.

One of the ways we calculate the error for the output for each observation is using the squared error function.

$$Error = \frac{1}{2}[actual\ value - predicted\ value]^2$$

$$\mathbf{E_{total}} = \frac{\mathbf{1}}{\mathbf{2}}[\mathbf{output} - \mathbf{o_{out}}]^{\mathbf{2}} \cdots (8)$$
$$E_{total} = \frac{1}{2}[0.01 - 0.75136507]^2 = 0.274811083$$
$$\text{where } 0.01 \text{ is the actual output and } 0.75136507 \text{ is the output predicted in the forward pass.}$$

Note: Here we have a single neuron in the output layer. In case there are multiple neurons, we sum the errors computed for each neuron on the output layer.

Backward Pass

Backpropagation is done to optimize the weights so that the neural network can learn how to get the optimum weights which result in minimum error.

Let us start with one of the weight parameters; consider $\mathbf{w_1'}$. We want to know how much a small change in w_1' will affect the total error E_{total}. In other words, we need to find the partial derivative of E_{total} w.r.t w_1' or the gradient w.r.t. w_1' i.e.$(\frac{\partial \mathbf{E_{total}}}{\partial \mathbf{w_1'}})$.

In order to find this, we will apply the chain rule from our knowledge of derivatives.

Applying the chain rule, we know:

$$\frac{\partial \mathbf{E_{total}}}{\partial \mathbf{w_1'}} = \frac{\partial \mathbf{E_{total}}}{\partial \mathbf{o_{out}}} * \frac{\partial \mathbf{o_{out}}}{\partial \mathbf{o_{in}}} * \frac{\partial \mathbf{o_{in}}}{\partial \mathbf{w_1'}} \cdots (9)$$

Lets compute each of these partial derivatives separately:

For the first term $\frac{\partial E_{total}}{\partial o_{out}}$, we know that:

$E_{total} = \frac{1}{2}[output - o_{out}]^2 \cdots (from\ (8))$

Taking derivative w.r.t. o_{out}???

$\frac{\partial E_{total}}{\partial o_{out}} = 2 * \frac{1}{2} * [output - o_{out}]^{(2-1)} * (-1) = -(output - o_{out}) = -(0.01 - 0.75136507) = 0.74136507 \cdots (10)$

For the second term $\frac{\partial o_{out}}{\partial o_{in}}$???, we know that:

$$o_{out} = \frac{1}{1+e^{-o_{in}}} \cdots (from\ (7))$$

Taking the derivative of a sigmoid function w.r.t. o_{in}

$$\frac{\partial o_{out}}{\partial o_{in}} = o_{out} * (1 - o_{out}) = 0.75136507 * (1 - 0.75136507) = 0.186815602 \cdots (11)$$

For the third term, we know that:

$$o_{in} = w'_0 * 1 + w'_1 * h_1 + w'_2 * h_2 \cdots (from\ (6))$$

Taking its derivative w.r.t. w'_1???

$$\frac{\partial o_{in}}{\partial w'_1} = h_1 * (w'_1)^{1-1} = h_1 = 0.593269992 \cdots (12)$$

Putting it all together:

$$\frac{\partial E_{total}}{\partial w'_1} = \frac{\partial E_{total}}{\partial o_{out}} * \frac{\partial o_{out}}{\partial o_{in}} * \frac{\partial o_{in}}{\partial w'_1}$$

$$\frac{\partial E_{total}}{\partial w'_1} = 0.74136507 * 0.186815602 * 0.593269992 \cdots (from (10), (11), (12))$$

$$\mathbf{\frac{\partial E_{total}}{\partial w'_1} = 0.082167041 \cdots (13)}$$

Once we have the value of $\frac{\partial E_{total}}{\partial w'_1}$ i.e. how much does the error changes with a small change in w'_1, we can now compute the new value of w'_1 as follows:

$$\mathbf{w'_{1new} = w'_1 - \eta * \frac{\partial E_{total}}{\partial w'_1}}$$

Here η??? is a special value known as the learning rate. Learning rate determines how quickly or slowly we want to update the parameters/weights. We will consider η??? as 0.5. This is a hyper-parameter to be tuned.

Putting in the numbers, we get the following value for w'_{1new}???:

$$w'_{1new} = w'_1 - \eta * \frac{\partial E_{total}}{\partial w'_1} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

0.35891648 will be the new updated weight for w'_1???.

Similarly, we can compute the value of w'_{2new}:

$$w'_{2new} = 0.408666186$$

Continuing the backward pass to the previous layer

We will continue with our backward pass to calculate the new values of w_{11}, w_{21}, w_{12}, and w_{22} as well.

Lets start with finding the updated value for w_{11}???. Similarly we will be able to find the values for w_{21}???, w_{12}???, and w_{22}??? too.

Using the chain rule, we know:

$$\frac{\partial E_{total}}{\partial w_{11}} = \frac{\partial E_{total}}{\partial h_1} * \frac{\partial h_1}{\partial s_1} * \frac{\partial s_1}{\partial w_{11}} \cdots (14)$$

Here again, we will compute each of the terms separately.

The first term $\frac{\partial E_{total}}{\partial h_1}$ can be computed as follows:

$$\frac{\partial E_{total}}{\partial h_1} = \frac{\partial E_{total}}{\partial o_{in}} * \frac{\partial o_{in}}{\partial h_1} \cdots (15)$$

Let us consider the term $\frac{\partial E_{total}}{\partial o_{in}}$ in the above equation (15):

$$\frac{\partial E_{total}}{\partial o_{in}} = \frac{\partial E_{total}}{\partial o_{out}} * \frac{\partial o_{out}}{\partial o_{in}}$$

We have computed both these values before; hence substituting we get the following:

$$\frac{\partial E_{total}}{\partial o_{in}} = \frac{\partial E_{total}}{\partial o_{out}} * \frac{\partial o_{out}}{\partial o_{in}}$$

$$\frac{\partial E_{total}}{\partial o_{in}} = 0.74136507 * 0.186815602 = 0.138498562 \cdots (\text{from (10) and (11)})$$

$$\frac{\partial E_{total}}{\partial o_{in}} = 0.138498562 \cdots (16)$$

Now we need to compute $\frac{\partial o_{in}}{\partial h_1}$??? in order to to compute the equation (15) i.e. in order to compute $\frac{\partial E_{total}}{\partial h_1}$???

We know that

$$o_{in} = w_0' * 1 + w_1' * h_1 + w_2' * h_2 \cdots (from(6))$$

Taking its derivative w.r.t. h_1??? gives:

$$\frac{\partial o_{in}}{\partial h_1} = w_1' = 0.4 \cdots (17)$$

Now we can compute $\frac{\partial E_{total}}{\partial h_1}$ substituting the values in equation (15):

$$\frac{\partial E_{total}}{\partial h_1} = 0.138498562 * 0.4 \cdots (from(16) and (17))$$

$$\frac{\partial E_{total}}{\partial h_1} = 0.055399425 \cdots (18)$$

Now, referring to equation (14) we still need to find the values of $\frac{\partial h_1}{\partial s_1}$ and $\frac{\partial s_1}{\partial w_{11}}$ to compute $\frac{\partial E_{total}}{\partial w_{11}}$???

In order to compute $\frac{\partial h_1}{\partial s_1}$???, we know that:

$$h_1 = \sigma(s_1) = \frac{1}{1 + e^{-s_1}} = 0.593269992 \cdots (from(4))$$

Taking its derivative w.r.t. s_1

$$\frac{\partial h_1}{\partial s_1} = h_1 * (1 - h_1)$$

$$\frac{\partial h_1}{\partial s_1} = 0.593269992 * (1 - 0.593269992)$$

$$\frac{\partial h_1}{\partial s_1} = 0.241300709 \cdots (19)$$

In order to compute $\frac{\partial s_1}{\partial w_{11}}$, we know that:

$$s_1 = w_{01} * 1 + w_{11} * x_1 + w_{21} * x_2 \cdots (\text{from (3)})$$

Taking the derivative of s_1 w.r.t. w_{11}

$$\frac{\partial s_1}{\partial w_{11}} = x_1 = 0.05 \cdots (20)$$

Putting it all together, we will be able to observe the effect of a small change in w_{11} on the overall error E_{total} i.e. $\frac{\partial E_{total}}{\partial w_{11}}$.

$$\frac{\partial E_{total}}{\partial w_{11}} = \frac{\partial E_{total}}{\partial h_1} * \frac{\partial h_1}{\partial s_1} * \frac{\partial s_1}{\partial w_{11}}$$

$$\frac{\partial E_{total}}{\partial w_{11}} = 0.055399425 * 0.241300709 * 0.05 \cdots (\text{from } (18),(19) \text{ and } (20))$$

$$\frac{\partial E_{total}}{\partial w_{11}} = 0.000438568$$

We can compute the new value for w_{11} as follows:

Putting in the numbers, we get the following value for w_{11new}:

$$w_{11new} = w_{11} - \eta * \frac{\partial E_{total}}{\partial w_{11}} = 0.15 - 0.5 * 0.000438568 = 0.149780716$$

Similarly we can compute the updated values for w_{12}, w_{21}, w_{22} as well.

This is how the weights/parameters in the neural network are updated.

After the first round of back-propagation, a forward pass is done again and we compute the error now. Now the total error E_{total} should be lesser than the previous total error.

This cycle will continue till we have a minimum possible overall error.

Reference: https://mattmazur.com/2015/03/17/a-step-by-step-back-propagation-example/

Summary of how neural networks are trained

In the forward pass, given inputs and random weights we understand how the output is computed. After the training of the neural network is complete, we run the forward pass only to make predictions.

But in order to be able to make these predictions that are close to the actual values, we need optimal weights. For this we train our model to learn the weights.

The training procedure works as follows:

- Initialize the weights for all the nodes randomly.
- For each observation in the training dataset, perform a forward pass using the current weights and calculate the output of each node from left to right. The final outcome is the output from the last node.
- We now compare the final output with the actual value in the training data. We now measure the error using a loss function for each observation.
- Using back-propagation, propagate the error to each individual node from right to left. Compute the contribution of each parameter/weight to the error and adjust the weights accordingly using, say gradient descent. Propagate the error gradients back starting from the last layer.

Reference: https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6

Neural Network Implementation using scikit learn

We will work with the census income dataset where we predict the income level i.e. whether it is more than 50K dollars or less than that amount given the characteristics of the people.

```
import pandas as pd
import numpy as np
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import RandomizedSearchCV
```

```
file=r'/Users/anjal/Dropbox/0.0 Data/census_income.csv'

cd= pd.read_csv(file)
```

```
cd['Y'].unique()
```

```
array([' <=50K', ' >50K'], dtype=object)
```

```
cd['Y']=(cd['Y']==' >50K').astype(int)
```

```
del cd['education']
# we have already discussed the reason for this earlier
```

```
cat_cols=cd.select_dtypes(['object']).columns
```

```
for col in cat_cols:
    freqs=cd[col].value_counts()
    k=freqs.index[freqs>99][:-1]
    for cat in k:
        name=col+'_'+cat
        cd[name]=(cd[col]==cat).astype(int)
    del cd[col]
```

```
cd.shape
```

```
(32561, 49)
```

After completing the data preparation part, we observe that there are about 32000 observations and 49 variables.

```
x_train=cd.drop(['Y'],axis=1)
y_train=cd['Y']
```

Parameters

- learning_rate: {'constant', 'invscaling', 'adaptive'}, default 'constant'
    - 'constant' is a constant learning rate given by the parameter 'learning_rate_init'
    - 'invscaling' gradually decreases the learning rate at each time step using an inverse scaling
    - 'adaptive' keeps the learning rate constant to 'learning_rate_init' as long as training loss keeps decreasing. Each time two consecutive epochs fail to decrease training loss by at least tol, or fail to increase validation score by at least tol if'early_stopping' is on, the current learning rate is divided by 5.
      Using 'inscaling' and 'adaptive' learning rates ensures that the parameter updates do not become infinitely slow as we reach towards the optimal point.
- hidden_layer_sizes : tuple, length = n_layers - 2, default (100,)
  (5,10,5) tells that there are three layers - first hidden layer has 5 nodes, second 10 nodes and third 5 nodes. This does not control the input and output layers.
  (20, 10) tells that there are two layers - the first layer has 20 neurons and the second layer has 10 neurons and so on
- alpha : L2 penalty (regularization term) parameter, default 0.0001

  This is a penalty term to our cost function to regularize the cost function so that we do not end up over optimizing the parameters. Higher the value of alpha, the more penalty we add.
- activation : {'identity', 'logistic', 'tanh', 'relu'}, default 'relu'
  Activation function for the hidden layer; this introduces non-linearity in the network
    - 'identity', useful to implement linear bottleneck,returns f(x) = x
    - 'logistic', the logistic sigmoid function, returns f(x) = 1 / (1 + exp(-x))
    - 'tanh', the hyperbolic tan function,returns f(x) = tanh(x).
    - 'relu', the rectified linear unit function,returns f(x) = max(0, x)

```
parameters={
'learning_rate': ["constant", "invscaling", "adaptive"],
'hidden_layer_sizes': [(5,10,5),(20,10),(10,20)],
'alpha': [0.3,.1,.01],
'activation': ["logistic", "relu", "tanh"]
}
```

We use Randomized Search to find which of these parameters give the best results.

```
clf=MLPClassifier()
```

```
random_search=RandomizedSearchCV(clf,n_iter=5,cv=10,param_distributions=parameters,

scoring='roc_auc',random_state=2,n_jobs=-1,verbose=False)
```

```
random_search.fit(x_train,y_train)
```

We get the model resulting from the best parameter combination using the following code:

```
random_search.best_estimator_
```

```
MLPClassifier(activation='relu', alpha=0.3, batch_size='auto', beta_1=0.9,
       beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=(5, 10, 5), learning_rate='adaptive',
       learning_rate_init=0.001, max_iter=200, momentum=0.9,
       n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
       random_state=None, shuffle=True, solver='adam', tol=0.0001,
       validation_fraction=0.1, verbose=False, warm_start=False)
```

```
def report(results, n_top=3):
    for i in range(1, n_top + 1):
        # np.flatnonzero extracts index of `True` in a boolean array
        candidate = np.flatnonzero(results['rank_test_score'] == i)[0]
        # print rank of the model
        # values passed to function format here are put in the curly brackets when printing
        # 0 , 1 etc refer to placeholder for position of values passed to format function
        # .3f means upto 2 decimal digits
        print("Model with rank: {0}".format(i))
        # this prints cross validate performance and its standard deviation
        print("Mean validation score: {0:.5f} (std: {1:.5f})".format(
              results['mean_test_score'][candidate],
              results['std_test_score'][candidate]))
        # prints the paramter combination for which this performance was obtained
        print("Parameters: {0}".format(results['params'][candidate]))
        print("")
```

```
report(random_search.cv_results_,5)
```

> Model with rank: 1
> Mean validation score: 0.61450 (std: 0.09325)
> Parameters: {'learning_rate': 'adaptive', 'hidden_layer_sizes': (5, 10, 5), 'alpha': 0.3, 'activation': 'relu'}

> Model with rank: 2
> Mean validation score: 0.60761 (std: 0.06540)
> Parameters: {'learning_rate': 'constant', 'hidden_layer_sizes': (20, 10), 'alpha': 0.3, 'activation': 'relu'}

> Model with rank: 3
> Mean validation score: 0.51063 (std: 0.01143)
> Parameters: {'learning_rate': 'adaptive', 'hidden_layer_sizes': (5, 10, 5), 'alpha': 0.01, 'activation': 'tanh'}

> Model with rank: 4
> Mean validation score: 0.50745 (std: 0.00640)
> Parameters: {'learning_rate': 'constant', 'hidden_layer_sizes': (10, 20), 'alpha': 0.01, 'activation': 'tanh'}

> Model with rank: 5
> Mean validation score: 0.50031 (std: 0.00227)
> Parameters: {'learning_rate': 'invscaling', 'hidden_layer_sizes': (10, 20), 'alpha': 0.01, 'activation': 'logistic'}

In the model giving the best performance, there are two hidden layers, the first layer having 20 nodes and the second layer having 10 nodes; regularization parameter is 0.3 and activation function is relu.

The mean validation score is 0.67559 for the selected parameters.

We have tried very few combinations. We can go ahead and try more parameter combinations and see if we get a better model performance.

```
mlp = random_search.best_estimator_
```

Now since we know the best parameter combination, we can fit the model on the entire training data and make predictions on the validation data.

```
mlp.fit(x_train,y_train)
```

After fitting the model, we make predictions just as we made for the algorithms discussed earlier.

Chapter 9 : Unsupervised Learning

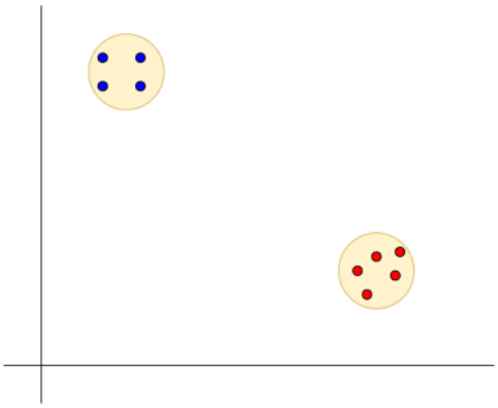
In this module, we are going to look at certain kind of business problems which are very different from problems where eventual agenda is to make predictions. Consider some of these problem statements :

- Finding segments of customer in your customer population which are different from each other and similar within to make more customized marketing campaigns
- Finding anomalies in the data, observations which are very different from the rest of the population
- Reduce representation redundancy in the data, decrease number of columns in the data without loss of information
- Visualize high dimensional data in 2D or 3D plane

None of these problems fit in the framework of predictive modelling that we have studied so far. There is no target to build the model for . However these are legitimate business problems to solve. We'll be learning techniques to address this kind of business problems here .

We'll start with addressing the first problem of finding groups which are similar within and different from each other . To address this problem we first need to define what do we mean by `similar`

Similarity Measures



In this simple image we have created two groups on the basis of our intuition. This intuition that we applied was nothing but on the basis of distances. We grouped the points which were closer together . Using euclidean distance as similarity measure is pretty common and works most of the data that we get to work with.

Distances

Euclidean distance however is a specific case of minkowski distances , which is defined as [between two points $P_1(x_1, x_2, x_3 \cdots, x_i \cdots)$ and $P_1(x_1', x_2', x_3' \cdots, x_i' \cdots)$] :

$$\left(\sum_i |x_i - x_i'|^p\right)^{1/p}, \text{ where } p \geq 1$$

- This becomes euclidean distance for p=2

$$\sqrt{\left(\sum_i |x_i - x_i'|^2\right)}$$

- for p=1 , its known as Manhattan distance

$$\left(\sum_i |x_i - x_i'|\right)$$

This could be a good choice for distance when all variables are not on same scale. Euclidean distances if used end up exaggerating difference in the dimension which is bigger on numeric scale.

Cosine Similarity

Cosine similarity is defined as the cosine of angle between two data points represented as vectors

$$cos(\theta) = \frac{\sum_{i=1}^{n} x_i * {x_i}'}{|\sqrt{\sum {x_i}^2}||\sqrt{\sum {x_i}'^2}|}$$

This takes values between `-1 to 1` , where `-1` implies most dissimilar and `1` means identical . This as similarity measure, is mostly used in context of text documents or categorical data.

Jaccard Similarity

This is used in context of sets and is defined as

$$\text{Jaccard Similarity} = \frac{|A \cap B|}{|A \cup B|}$$

Where $|S|$ for set is number of unique elements in the set. [Also known as cardinality of the set]. This again is used mostly in context of text data or categorical data.

All these different distances/similarity measures can be computed for a pair of points. But we also need to come up with some way of calculating distances between the groups of points . Why ? Imagine how will you arrive at say 3 groups , for a datasets containing 1000 observations .

Data Standardisation

Distances which rely on scale of the individual dimension might end up exaggerating impact of some dimensions over the other . Here is an example to understand this intuitively , Consider these two individuals with their age and income given :

$$P_1 = (Age = 23, Income = \text{Rs } 6{,}10{,}000)$$
$$P_2 = (Age = 43, Income = \text{Rs } 6{,}00{,}000)$$

Intuitively we know that, the difference in ages here is much noticeable in comparison to the difference between their income. And this intuition is driven by the natural scale of ages , where a difference of 20 years is significant whereas a difference of just Rs 10,000 is not much in context of income. However if we let this data be at its given scale and calculate the distance , see what happens

$$Dist(P_1, P_2) = \sqrt{20^2 + 10000^2} \approx 10000.02$$

Because of Age being on smaller numeric scale , difference there doesn't matter at all . This can be mitigated by removing effect of scale . This is achieved by centering the data with mean [or median] and scaling the data with standard deviation [or MAD or RANGE etc] .

Here is mathematical explanation of the process and impact it has . Lets say there is one column X which takes values $\{x_1, x_2, x_3 \cdots x_n\}$

We transform this into a new column Y such that :

$$y_i = \frac{x_i - \overline{X}}{S_X}$$

Where $\overline{X}$ is mean of column X and S_X is standard deviation of column X, defined as :

$$\overline{X} = \frac{\sum_{i=1}^{n} x_i}{n}$$
$$S_X = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \overline{X})^2}{n}}$$

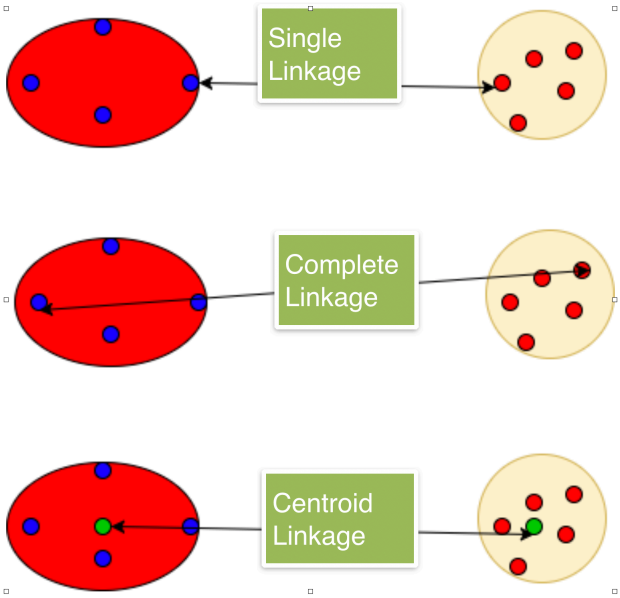
Now lets see what is the mean and standard deviation of this standardized version of X [that is Y].

$$
\begin{aligned}
\overline{Y} &= \frac{\sum_{i=1}^{n} y_i}{n} \\
&= \frac{\sum_{i=1}^{n} (x_i - \overline{X})}{nS_X} \\
&= \frac{\sum x_i - \sum \overline{X}}{nS_x} \\
&= \frac{n\overline{X} - n\overline{X}}{nS_x} \\
&= 0
\end{aligned}
$$

$$
\begin{aligned}
S_Y &= \sqrt{\frac{\sum (y_i - \overline{Y})^2}{n}} \\
&= \sqrt{\frac{\sum {y_i}^2}{n}} \qquad since\ \overline{Y} = 0 \\
&= \sqrt{\frac{\sum \left(\frac{x_i - \overline{X}}{S_X}\right)^2}{n}} \\
&= \frac{1}{S_X} \sqrt{\frac{\sum_{i=1}^{n} (x_i - \overline{X})^2}{n}} \\
&= \frac{S_X}{S_X} \\
&= 1
\end{aligned}
$$

This means that, the transformed column will have mean 0 and standard deviation 1 , irrespective of what values the column takes . This can bring all the columns to same scale and then remove the effect of larger numeric scales on distance calculations. Impact of using other measures for centering and scaling is also similar if not exactly same .

Distance Between the Groups [Linkage Methods]



- **Single Linkage** : In this method, distance between two groups is defined as distance between points of the groups which are nearest to each other. This method leads to groups which are as dispersed as possible.

- **Complete Linkage**: Here, distance between two groups is defined as distance between points of the groups which are farthest from each other. This method leads to groups which are as compact as possible
- **Centroid Linkage** : Here, the distance between the groups is defined as simple distance between centroid of the groups. This is computationally much less expensive and one of the most popular methods to calculate distances between the groups.

  There are some other notable methods : `Average Linkage` , `Ward's Method`

Hierarchical Clustering (Agglomerative Clustering)

Let's take an example of grouping points , let's say we are given distance between points already and we are going to use complete linkage method to find distance between groups . There are five points A,B,C,D,E and distance matrix for them looks like this :

-	A	B	C	D	E
A	X	2	7	9	10
B	-	X	1	8	6
C	-	-	X	5	4
D	-	-	-	X	3
E	-	-	-	-	X

The lower half of the matrix is left empty because it'll take same values . Given this matrix , we can see that the closest points are B & C. Those will be the first to get combined . Once they are in the group , new distance matrix will look like this :

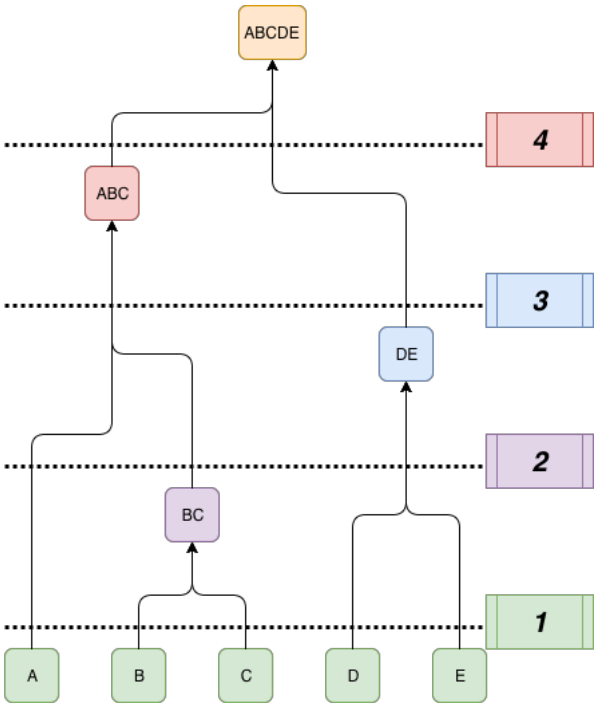
-	A	BC	D	E
A	X	7	9	10
BC	-	X	8	6
D	-	-	X	3
E	-	-	-	X

Here, distance between group BC and other points is calculated using complete linkage method. The longest distance. **Remember that joining of points and groups will always be on the basis of minimum distance.** Complete linkage method is just a way of *calculating* the distances.

Now minimum distance is between points D and E, those will be the next to get combined .

-	A	BC	DE
A	X	7	10
BC	-	X	8
D	-	-	X

Next to be combined , will be A and BC, this will leave us with only two groups which will the ultimate ones to be joined . If we visualize this process, it looks like this .



This is known as a dendrogram, and the process that we just carried out is called hierarchical clustering . In the process if we stopped at `1` , we'll have no groups , all the points are separate. If we stopped at `2` , we'll have 4 groups : $\{A\}, \{B, C\}, \{D\}, \{E\}$

If we stopped at `3` , we'll have 3 groups : $\{A\}, \{B, C\}, \{D, E\}$

If we stopped at `4` we'll have two groups $\{A, B, C\}, \{D, E\}$

In the next stop, we'll be left with just one group with all the points in it . You can see that this process computationally is pretty complex and this complexity increases exponentially as number of data points go up. It's not a very popular clustering algorithm for the same reason. We'll be discussing the alternative named K-means clustering once we are through the implementation of Hierarchical clustering in python using `sk-learn`

Agglomerative clustering in Python

`Problem Statement` : We have been given data on chemical properties of wine . Our job is to find groups of wines on the basis of their potency and sulphate content (acts as antioxidants).

Note : In general what variables should be considered during the clustering process; is a business process mandate. There are no statistical or mathematical measures to tell you that one variables is bad and another is not [unlike predictive modelling problems where we had a definite target]

In this case our variables of interest are `sulphates` and `alcohol` . Apart from imports , we'll standardize the data before start the clustering process as discussed above .

```
myfile=r'/Users/lalitsachan/Dropbox/0.0 Data/winequality-white.csv'

import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns

from sklearn.preprocessing import scale
from sklearn.cluster import KMeans
from sklearn.metrics import  silhouette_score

%matplotlib inline

wine=pd.read_csv(myfile,sep=";")
```

```
wine=wine[["sulphates","alcohol"]]
wine.head()
```

	sulphates	alcohol
0	0.45	8.8
1	0.49	9.5
2	0.44	10.1
3	0.40	9.9
4	0.40	9.9

We can see that these two variables differ a lot on numeric scale , let's have a quick look at their comparative mean and standard deviation .

```
wine.agg(['mean','std'])
```

	sulphates	alcohol
mean	0.489847	10.514267
std	0.114126	1.230621

Let's standardize the data .

```
from sklearn.preprocessing import StandardScaler
st=StandardScaler()
wine_std=pd.DataFrame(st.fit_transform(wine),columns=list(wine.columns))
wine_std.agg(['mean','std'])
```

	sulphates	alcohol
mean	-7.572999e-16	-2.178784e-15
std	1.000102e+00	1.000102e+00

We can see now that the means are practically zero and standard deviations 1 for both the variables. We can start with clustering process. Before we go ahead , we need to figure out a way to decide how many groups/cluster is an optimal choice for the data which we have been given .

Measuring Goodness of Clustering Results : Silhouette Score

For each Observation i, let a_i be the average distance between i and all other data within the same cluster. We can interpret a_i as a measure of how well i is assigned to its cluster (the smaller the value, the better the assignment). We then define the average dissimilarity of point i to a cluster c as the average of the distance from i to all points in c.

Let b_i be the smallest average distance of i to all points in any other cluster, of which i is not a member. The cluster with this smallest average dissimilarity is said to be the "neighboring cluster" of i because it is the next best fit cluster for point i.

We now define a silhouette:

$$s_i = \frac{b_i - a_i}{\max\{a_i, b_i\}}$$

Which can be also written as:

$$s_i = \begin{cases} 1 - \frac{a_i}{b_i}, & \text{if } a_i < b_i \\ 0, & \text{if } a_i = b_i \\ \frac{b_i}{a_i} - 1, & \text{if } a_i > b_i \end{cases}$$

We can clearly see that $s_i \in [-1, 1]$, if $s_i \to -1$, it implies that the observation does not belong to cluster to which it has been assigned . If $s_i \to +1$, it means observation belongs to the cluster it has been assigned to. Silhouette Score for a cluster can be calculate as average of silhouette index values for individual observations in the cluster. Silhouette score for an entire data can be calculated as average of silhouette index values for all the data points.

We can carry out the clustering process with different number of clusters and eventually choose the one which has highest silhouette score . Let's now go ahead without clustering problem.

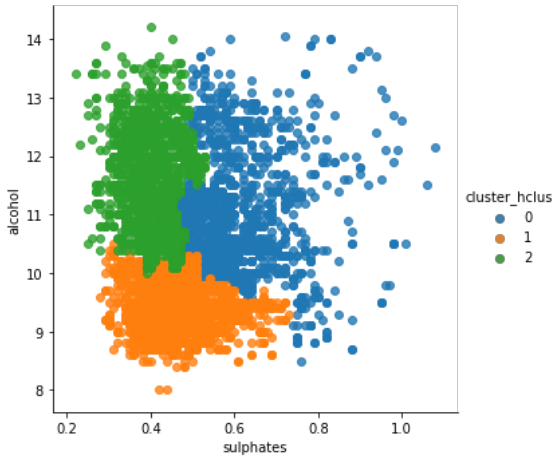
```
from sklearn.cluster import AgglomerativeClustering
```

```
for n_clusters in range(2,10):
    cluster_model = AgglomerativeClustering(n_clusters=n_clusters,
  affinity='euclidean',linkage='ward')
    cluster_labels = cluster_model.fit_predict(wine_std)
    silhouette_avg = silhouette_score(wine_std,cluster_labels,metric='euclidean')
    print("For n_clusters =", n_clusters,
          "The average silhouette_score is:", silhouette_avg)
```

```
For n_clusters = 2 The average silhouette_score is: 0.30910956895473835
For n_clusters = 3 The average silhouette_score is: 0.36507065689263013
For n_clusters = 4 The average silhouette_score is: 0.36394118955693916
For n_clusters = 5 The average silhouette_score is: 0.2849369402765109
For n_clusters = 6 The average silhouette_score is: 0.28173689619112985
For n_clusters = 7 The average silhouette_score is: 0.30584775050004287
For n_clusters = 8 The average silhouette_score is: 0.29918403546397854
For n_clusters = 9 The average silhouette_score is: 0.2931385252344828
```

We can see that `3` is the optimal cluster number . Let's go ahead with that and look at the results .

```
hclus=AgglomerativeClustering(n_clusters=3, affinity='euclidean',linkage='ward')
labels_hclus=hclus.fit_predict(wine_std)
wine['cluster_hclus']=labels_hclus
sns.lmplot(fit_reg=False,x='sulphates',y='alcohol',data=wine,hue='cluster_hclus')
```

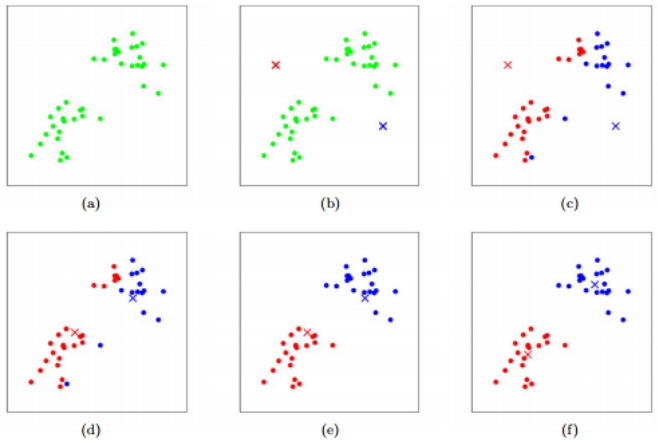


Apart from visualization you can also , look at cluster wise numerical summaries to see how the clusters/groups differ from each other . In fact when you have number of variables , you wont be able to visualize clusters like this , numerical summaries is all that you'll have . We'll see in this module, in some time , one way to visualize high dimensional data in 2D-space.[known as `t-sne`]

Note : Cluster numbers 0,1,2 as such are not ordinal . Only the membership in the clustering matters .

K-Means Clustering

Hierarchical clustering requires distance matrix calculation between all the points. Which comes out to be an expensive process computationally. We need an algorithm which scales well computationally with number of observations in the data. K-means is such an algorithm. Algorithm needs number of clusters as input. For example, in the figure given below , we can see that in (a) there are two groups present in the data. We'll see if K-means can group these points properly . Here is how the algorithm worked :



(b) $\rightarrow$ Algorithm randomly selects two points and considers them to be group centers

(c) $\rightarrow$ Rest of the points in the data are assigned membership to the two groups on the basis of distances from these group centers

This of-course is not a proper grouping. Process doesn't stop here . This was the first iteration .

(d) $\rightarrow$ New group centers are calculated on the basis of the previous groups formed.

(d) $\rightarrow$ Points are re-assigned membership on the basis of these new group centers

This process is carried out for multiple iterations, until either observation membership stops to change or new group centers are very close to the old ones . In very few iterations, as you can see observation's membership gravitates towards natural groups present in the data.

How do we, however come up with the value for K? Since K-means is cheaper algorithm computationally; we can run it for multiple values of K, and choose that value of K as best for which the silhouette score comes out to be the best.

```
range_n_clusters = [2, 3, 4, 5, 6,7,8,9
for k in range_n_clusters:
kmeans=KMeans(n_clusters=k)
kmeans.fit(wine_std)
print(k,silhouette_score(wine_std,kmeans.labels_))
```

> 2, 0.3739606160278498
> 3, 0.4108674619159248
> 4, 0.38344260989210777
> 5, 0.33457661751536383
> 6, 0.3477567148831231
> 7, 0.3519661585228609
> 8, 0.35398240541646087
> 9, 0.3507188899680227

We can again see that `3` is the optimal value

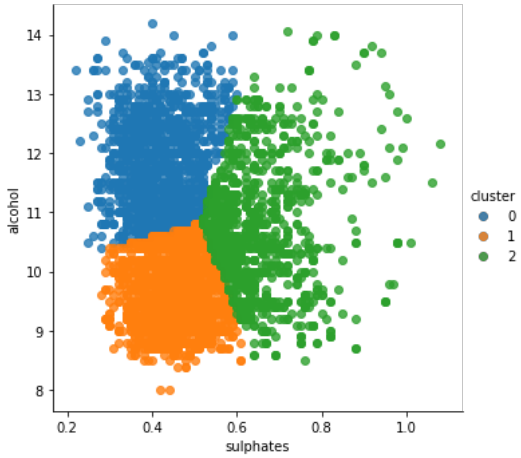
```
k = 3
kmeans = KMeans(n_clusters=k)
kmeans.fit(wine_std)
labels = kmeans.labels_
wine["cluster"]=labels
```

Number of observations in each cluster

```
wine['cluster'].value_counts
```

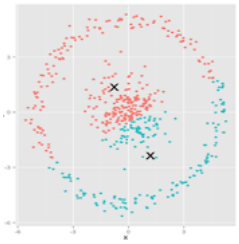
```
1  2281
0  1476
2  1141
```

```
sns.lmplot(fit_reg=False,x='sulphates',y='alcohol',data=wine,hue='cluster')
```

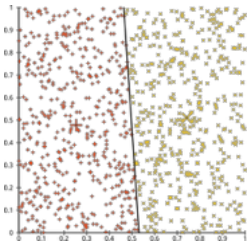


Issues with K-Means

- If the shape of the groups is spherical , it ends up making nonsensical groups in the data



- It doesn't care, whether there really do exist different groups in the data. It will forcefully separate data into groups as per the value of K, even if there do not exist any natural groups in the data



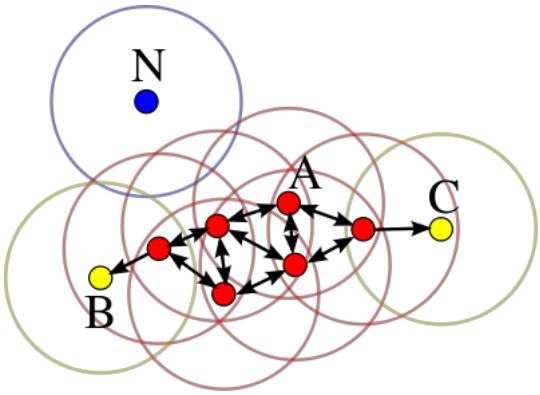
- It is susceptible to extreme values in the data and might give rise to improper grouping if such is the data that we are dealing with.

DBSCAN (Density-based spatial clustering of applications with noise)

This particular algorithm addresses issue associate with K-means discussed above . Basic idea behind DBSCAN is that

- Clusters are dense regions in the data space, separated by regions of lower object density
- A cluster is defined as a maximal set of density- connected points
- Discovers clusters of arbitrary shape

It has two parameters ϵ and `minPts` , ϵ represent the neighborhood size and minPts represents the minimum number of points required for a point to be core points. Let's see how DBSCAN actually does clustering . It starts with randomly selecting a point in the data, next action depends on what kind of point it is .



In this diagram, minPts = 4. Point A and the other red points are core points, because the area surrounding these points in an ϵ radius contain at least 4 points (including the point itself). Because they are all reachable from one another, they form a single cluster. Points B and C are not core points, but are reachable from A (via other core points) and thus belong to the cluster as well. Point N is a noise point that is neither a core point nor directly-reachable.

- Algorithm starts by selecting a random point
- If its Noise point, it selects another point [and keeps on going until it finds a non-noise point]
- After selecting a point, it looks at other points in the neighborhood [defined by ϵ] and assigns them in the same group .
- Among these recent members of the group, non-core points are not processed further. Algorithm looks in the neighborhood of these new core points and assigns all the points to the same group
- Above process is carried out, until there are no points remaining in the neighborhood of core points found so far
- Algorithm again selects a random points from the so far un-assigned points in the data

$\rightarrow$ Noise points are the points which are assigned to no groups and can be labelled as anomalies

$\rightarrow$ Due to progression of membership assignment as explained above , arbitrary shape groups can be captured

$\rightarrow$ Algorithm doesn't need any input for number of groups , it finds as many groups as there are naturally present [This however can be varied by controlling values of ϵ]

This source provides nice animation of the process to better understand how DBSCAN works :

https://www.naftaliharris.com/blog/visualizing-dbscan-clustering/

DBSCAN in python

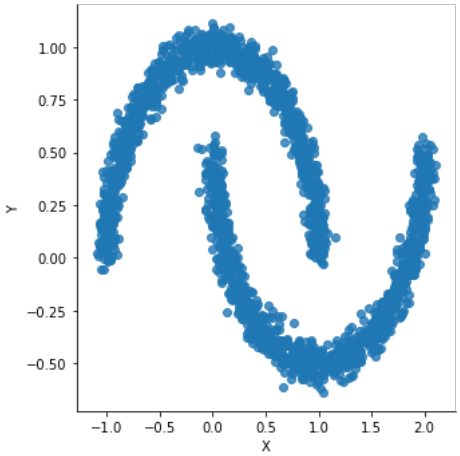
we'll work with an example which will give you a comparison between K-means and DBSCAN results as well .

```
mydata=pd.read_csv("/Users/lalitsachan/Dropbox/0.0 Data/moon_data.csv").iloc[:,1:]
mydata.head()
```

	X	Y
0	1.045255	0.332214
1	0.801944	-0.411547
2	-0.749356	0.775108
3	0.975674	0.191768
4	-0.512188	0.929997

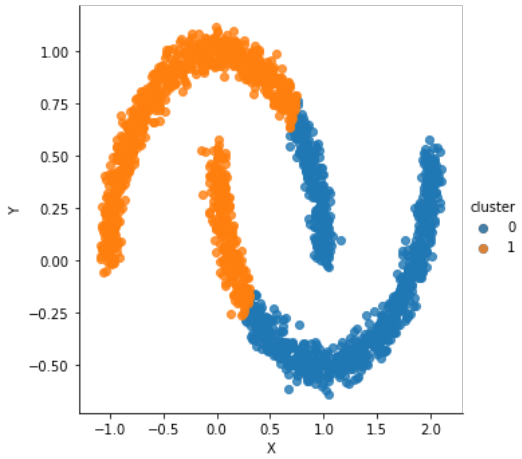
Data is already standardized, we can go for the clustering directly . Lets see how the data looks like .

```
sns.lmplot('X','Y',data=mydata,fit_reg=False)
```



We can clearly see that there are two groups in the data, however they are not spherical in nature . Lets see how K-means does on this and then we'll results from DBSCAN.

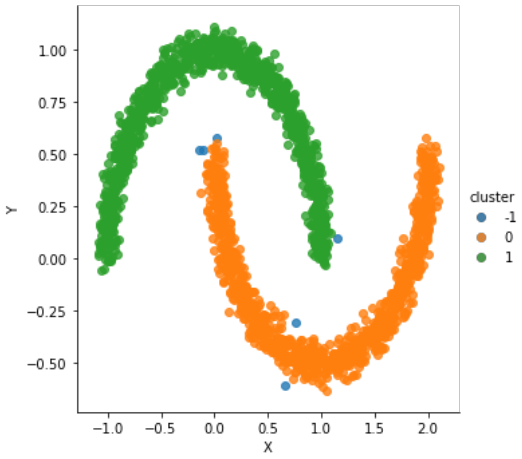
```
kmeans=KMeans(n_clusters=2)
kmeans.fit(mydata)
mydata["cluster"]=kmeans.labels_
sns.lmplot('X','Y',data=mydata,hue='cluster',fit_reg=False)
```



Lets remove the results from K-means and populate them with results from DBSCAN

```
del mydata['cluster']
```

```
from sklearn.cluster import DBSCAN
db = DBSCAN(eps=0.1, min_samples=30, metric='euclidean').fit(mydata)
mydata['cluster']=db.labels_
sns.lmplot('X','Y',data=mydata,hue='cluster',fit_reg=False)
```




```
pd.Series(db.labels_).value_counts()
```

```
1   999
0   995
-1    6
```

`-1` here are the points which are not part of any group and can be labelled as anomalies. Try playing around with different values of ϵ and `minPts` and see the impact .

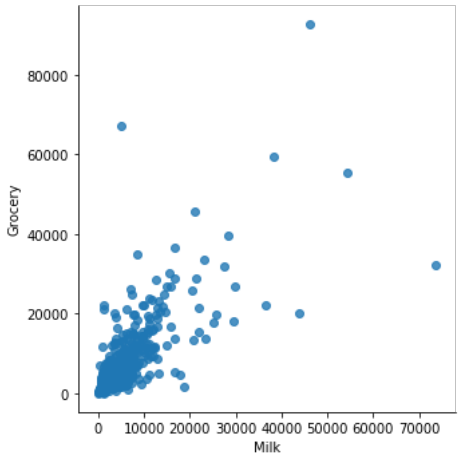
Anamoly detection with DBSCAN

We have been given consumption data for some retail customer. Our task is to mark 5% of the customers which are very different from rest of the population . This needs to be done with respect to two product categories `Milk` and `Groceries` .

```
myfile=r'/Users/lalitsachan/Dropbox/PDS V3/Data/Wholesale customers data.csv'
groc=pd.read_csv(myfile)
groc=groc[["Milk","Grocery"]]
groc_std=pd.DataFrame(scale(groc),columns=list(groc.columns))
```

Lets see how the data looks

```
sns.lmplot(x='Milk',y='Grocery',data=groc,fit_reg=False)
```



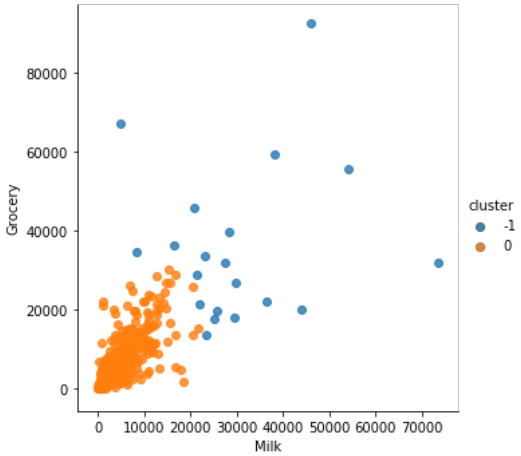
Lets try out different values of ϵ , and see how many values are labelled as outliers

```
r=np.linspace(0.5,4,10)
for epsilon in r:
    db = DBSCAN(eps=epsilon, min_samples=20, metric='euclidean').fit(groc_std)
    labels = db.labels_
    #n_clust=len(set(labels))-1
    outlier=np.round((labels == -1).sum()/len(labels)*100,2)
    #print('Estimated number of clusters: %d', n_clust)
    print("For epsilon =", np.round(epsilon) ,", percentage of outliers is:
",outlier)
```

```
For epsilon = 0.50 , percentage of outliers is:  9.77
For epsilon = 0.89 , percentage of outliers is:  4.55
For epsilon = 1.28 , percentage of outliers is:  2.05
For epsilon = 1.67 , percentage of outliers is:  1.59
For epsilon = 2.06 , percentage of outliers is:  1.36
For epsilon = 2.44 , percentage of outliers is:  1.14
For epsilon = 2.83 , percentage of outliers is:  0.91
For epsilon = 3.22 , percentage of outliers is:  0.68
For epsilon = 3.61 , percentage of outliers is:  0.68
For epsilon = 4.00 , percentage of outliers is:  0.45
```

we'll go with `0.89`

```
db = DBSCAN(eps=0.89, min_samples=20, metric='euclidean').fit(groc_std)
groc['cluster']=db.labels_
sns.lmplot(x='Milk',y='Grocery',data=groc,fit_reg=False,hue='cluster')
```

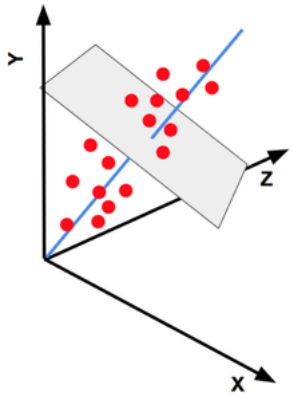


Dimensionality Reduction with PCA

Many at times, we work with data which has lot of redundancy in representation. Many variables/columns are highly correlated to each other . So much so that whole data can be represented with few number of columns which are nothing but linear combinations of previous columns; without any loss of information (variance in the data) .

This can be achieved by projecting the high dimensional data onto a lower dimension subspace. Lets understand what do we mean by that .

- We start with p dimensional data vectors (observations)
- Dimension is reduced by projecting them onto a q-dimensional (q < p) subspace
- This is done while preserving variance in the data



This figure here shows what do we mean by projection. Red dots represent data in 3 dimensions . Grey plane represents the 2 dimensional subspace . If we project these points on that plane, we'll have 2 dimensional representation of the data. The subspace however has the basis which is linear combination of 3 dimensional basis.

Mathematically if we project a vector $\overrightarrow{x_i}$ on to a subspace represented by a unit vector $\vec{w}$, its projection can be written as :

$$(\overrightarrow{x_i} \cdot \vec{w})\vec{w}$$

dot product represent the norm of the vector and $\vec{w}$ the spatial orientation , since that's the unit vector representing the spatial orientation of the plane onto which we are projecting the data. Now there are infinitely many possible 2D plane in the example above. Which 2D plane should we select to project our data on such that we get the lower dimensional representation with least amount of loss in variance of the data. The lower dimension subspace is essentially defined by the unit vector w .

So mathematically we want to find out **unit vector** w such that :

$$\sum_{i=1}^{n} ||\overrightarrow{x_i} - (\overrightarrow{x_i} \cdot \vec{w})\vec{w}||^2$$

is as small as possible . [That is the difference or loss in variance/information after projection]. One of the things which makes our lives mathematically easy that the data is centered before projection. which means

$$\sum_{i=1}^{n} \overrightarrow{x_i} = 0 \qquad \cdots (1)$$

Lets try to simplify our objective here :

$$\sum_{i=1}^{n} ||\overrightarrow{x_i} - (\overrightarrow{x_i} \cdot \vec{w})\vec{w}||^2$$

$$= \sum_{i=1}^{n} (\overrightarrow{x_i} \cdot \overrightarrow{x_i}) + ((\overrightarrow{x_i} \cdot \vec{w})\vec{w} \cdot (\overrightarrow{x_i} \cdot \vec{w})\vec{w}) - 2(\overrightarrow{x_i} \cdot \vec{w})\vec{w} \cdot \overrightarrow{x_i}$$

$$= \sum_{i=1}^{n} ||x_i||^2 + (\overrightarrow{x_i} \cdot \vec{w})^2 \vec{w} \cdot \vec{w} - 2(\overrightarrow{x_i} \cdot \vec{w})^2$$

$$= \sum_{i=1}^{n} ||x_i||^2 - (\overrightarrow{x_i} \cdot \vec{w})^2$$

$$\text{since } \vec{w} \text{ is a unit vector } \implies \vec{w} \cdot \vec{w} = 1$$

For a fixed given data $\sum_{i=1}^{n} ||\overrightarrow{x_i}||^2$ will be constant. This makes our objective to maximize the expression

$$\sum_{i=1}^{n} (\overrightarrow{x_i} \cdot \vec{w})^2$$

$$or$$

$$\frac{1}{n} \sum_{i=1}^{n} (\overrightarrow{x_i} \cdot \vec{w})^2$$

we can divide by `n`, it being a constant , doesn't change the result of objective optimization . We can use the following identity to further change the objective .

$$\text{Mean of Squares} = \text{Squares of the Mean} + \text{Variance}$$

$$\frac{1}{n} \sum_{i=1}^{n} (\overrightarrow{x_i} \cdot \vec{w})^2$$

$$= \Big[\frac{1}{n} \sum_{i=1}^{n} (\overrightarrow{x_i} \cdot \vec{w})\Big]^2 + var[\overrightarrow{x_i} \cdot \vec{w}]$$

$$= \Big[\frac{1}{n} (\sum_{i=1}^{n} \overrightarrow{x_i}) \cdot \vec{w})\Big]^2 + var[\overrightarrow{x_i} \cdot \vec{w}]$$

$$\text{using eq } (1) \text{ above}$$

$$= var[\overrightarrow{x_i} \cdot \vec{w}]$$

Now , the problem to solve here is :

$$\max\ var[\overrightarrow{x_i} \cdot \vec{w}]\ \ s.t. ||\vec{w}|| = 1$$

Same thing written in matrix format :

$$\max\ (XW)^T(XW)\ \ s.t. W^TW = 1$$

$$\max\ W^TX^TXW\ \ s.t. W^TW = 1$$

$$\max\ W^TVW\ \ s.t. W^TW = 1$$

$$[\text{where } X^TX \text{is written as V the variance covariance matrix for the given data}]$$

combining the constraints with the objective using Lagrange's multiplier, new objective becomes :

$$W^TVW - \lambda(W^TW - 1)$$

We need to maximize this w.r.t. W, lets differentiate this w.r.t. W and equate it to zero. We get :

$$V\vec{w} = \lambda\vec{w}$$

If you recall little bit of matrix algebra, this is a classic equation for eigen values. $\vec{w}$ here is nothing but eigen vectors of variance-covariance matrix of the given data and λ are eigen values.

- Variance-Covariance matrix is symmetric, its Eigen vectors are all orthogonal and distinct
- These are known as principal components

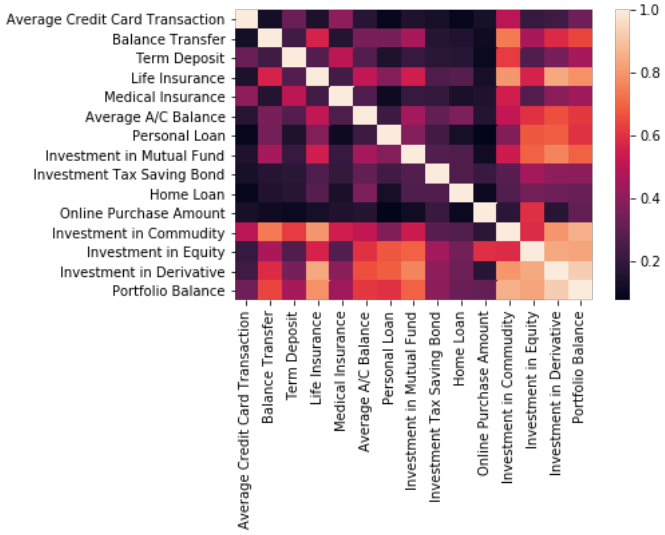
- These can be arranged in terms of variance explained along with direction of each principal component
- Number of principal components are same as number of variables , however we can always ignore the ones which do not contribute much towards explaining variance in the data; which will eventually result in dimensionality reduction

Lets look at an example in python :

```
import pandas as pd
import numpy as np
from sklearn.decomposition import PCA
from sklearn.preprocessing import scale

file=r'/Users/lalitsachan/Dropbox/PDS V3/Data/Existing Base.csv'
bd=pd.read_csv(file)

# selecting numeric columns which have redundancy in representation
bd=bd.select_dtypes(exclude=['object'])
bd.drop(['REF_NO','year_last_moved','Revenue Grid'],1,inplace=True)
```

we can visualize the correlation as follows :

```
import seaborn as sns
%matplotlib inline
sns.heatmap(bd.corr())
```



You can see that there are many variables [total 15 vars in the data] which are highly correlated to each other. We'll first scale the data and then apply pca with 15 components [same as number of vars in the data] , to see how many PCAs we can ignore on the basis of variance explained .

```
X=bd.copy()
X = scale(X)
pca = PCA(n_components=15)
pca.fit(X)
```

Lets look at variance explained by individual PCs

```
var= np.round(pca.explained_variance_ratio_,3)
print(var)
```

```
0.458 0.107 0.084 0.068 0.055 0.046 0.045 0.041 0.033 0.031 0.027 0.005 0.  0.  0.
```

You can see that last few PCs have almost zero contribution towards explained variance. lets look at cumulative explained variance as we increase number of PCs.

```
var1=np.cumsum(np.round(pca.explained_variance_ratio_, decimals=4)*100)
print(var1)
```

```
45.78  56.46  64.84  71.61  77.13  81.74  86.22  90.36  93.7   96.84  99.57.  100.02.  100.02.
100.02  100.02
```

You can see that by 10 or so PCs we have covered almost ~97% variance. This means that we can bring down the dimension of the data from 15 to 10 (reducing the data size by 33%) without much loss to information .

```
pca = PCA(n_components=10)
pca.fit(X)
X1=pd.DataFrame(pca.transform(X))
X1.head()
```

	0	1	2	3	4	5	6	7	8	9
0	0.869774	-0.792852	-0.300362	0.463627	0.016042	0.368796	-2.804879	0.659902	0.076150	0.546220
1	-1.763740	0.271868	-0.529384	0.342982	0.391251	-0.243377	-0.136108	0.514181	0.008195	-0.106689
2	2.759896	0.575463	-1.617375	-0.904037	2.031229	-0.561042	-0.315313	-0.826448	2.165835	-0.664759
3	-1.682247	0.607132	0.003920	-0.086971	0.092131	0.192318	0.067067	-0.697390	0.119397	0.299323
4	3.433527	0.456228	0.402857	-2.512890	0.613241	2.756212	0.126809	0.513503	-2.382016	-0.410804

We can use this new data with 10 columns , instead of the original one with 15. Each individual column here is linear combination of 15 columns of the original one. Here is how you can get loadings for any one of the columns, lets look at for the first one.

```
# pc1
loadings=pca.components_[0]
loadings
```

```
0.13851952, 0.25191454, 0.18841748, 0.30460754, 0.18251027,
0.2582609 , 0.22451307, 0.27721035, 0.17442024, 0.15276683,
0.12107231, 0.34070339, 0.32922577, 0.3650171 , 0.36834711
```

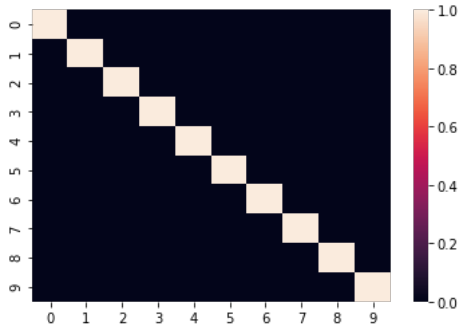
These number will make more sense , along with variables which they are loadings for

```
list(zip(bd.columns,loadings))
```

```
('Average Credit Card Transaction', 0.1385195225996486),
('Balance Transfer', 0.2519145395075939),
('Term Deposit', 0.18841747578001114),
('Life Insurance', 0.3046075395872803),
('Medical Insurance', 0.18251026979222082),
('Average A/C Balance', 0.25826089686660914),
('Personal Loan', 0.22451306738791554),
('Investment in Mutual Fund', 0.27721035413260525),
('Investment Tax Saving Bond', 0.17442024066219247),
('Home Loan', 0.15276682941684103),
('Online Purchase Amount', 0.1210723066277006),
('Investment in Commudity', 0.3407033948399679),
('Investment in Equity', 0.3292257708491981),
('Investment in Derivative', 0.365017101995759),
('Portfolio Balance', 0.3683471146126874)
```

Last thing to confirm the orthogonality [and in turn zero dependence] of the columns of this new data, lets look at correlation heat map again .

```
sns.heatmap(X1.corr())
```



you can see zero correlation among any of the new column.

Dimensionality Reduction for Visualising Multi-Dimensional data with t-SNE

We can technically use PCA to bring down the dimensionality arbitrarily. How ever being restricted to make use of just linear combinations , in absence of any correlation that forced dimensionality reduction, results in; not so true representation of the data. t-SNE, unlike PCA, is not a linear projection. It uses the **local relationships**between points to create a low-dimensional mapping. This allows it to capture **non-linear structure**.

How t-SNE works:

t-SNE ??? at a high level ??? basically works like this:

- Step 1: In the high-dimensional space, create a **probability distribution** that dictates the relationships between various neighboring points
- Step 2: It then tries to **recreate** a low dimensional space that follows that probability distribution as best as possible.

The ???t??? in t-SNE comes from the t-distribution, which is the distribution used in Step 2. The ??? S??? and ???N??? (???stochastic??? and ???neighbor???) come from the fact that it uses a probability distribution across neighboring points.

For more details on mathematical process, please refer to this : https://mlexplained.com/2018/09/14/paper-dissected-visualizing-data-using-t-sne-explained/

Lets see it in practice . Remember that we said it is impossible to visualized group separation that we get from clustering, because we are limited to 2D or at max 3D visualization . How about we make use of t-SNE here .

We'll use the same wine data but make use of all the variables and then visualize the resultant clustering .

```
wine=pd.read_csv(r'/Users/lalitsachan/Dropbox/0.0 Data/winequality-white.csv',sep=";")
X=scale(wine.iloc[:,:-1])
# ignoring the last column quality here
# using all other columns in the data
```

```
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
for k in range(2,10):
    kmeans=KMeans(n_clusters=k)
    kmeans.fit(X)

    print(k,silhouette_score(X,kmeans.labels_))
```

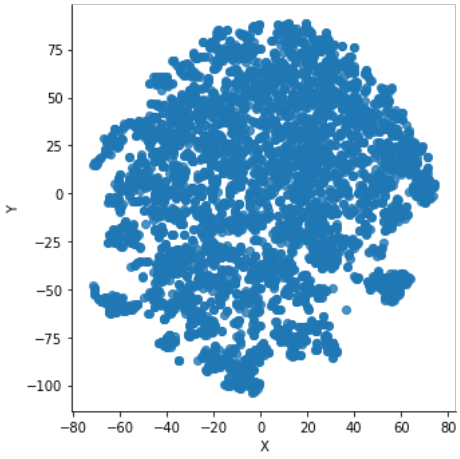
```
2 0.21447656060330086
3 0.14430589277155192
4 0.15898404552909834
5 0.14395682107234764
6 0.14588278206706412
7 0.12773278998393625
8 0.12828212706539913
9 0.12737121583698335
```

We can see that 2 seems to be the ideal number here .

```
kmeans=KMeans(n_clusters=2)
kmeans.fit(X)
X_2d=tsne.fit_transform(X)
mydata=pd.DataFrame(X_2d,columns=['X','Y'])
mydata['cluster']=kmeans.labels_
```

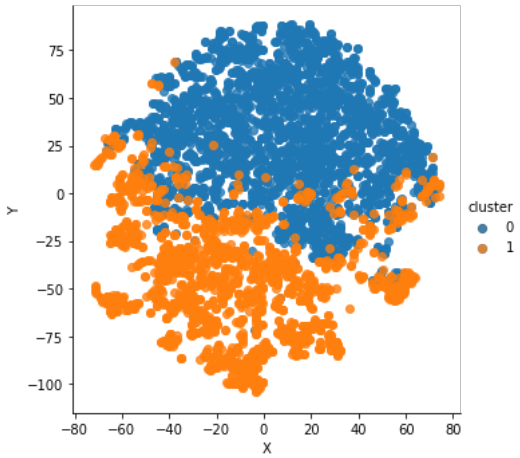
this is how the data, when transformed into 2D, looks like

```
sns.lmplot('X','Y',data=mydata,fit_reg=False)
```



Lets overlap the clustering results on it

```
sns.lmplot('X','Y',hue='cluster',data=mydata,fit_reg=False)
```



We can see the separation of the groups is largely fine. There will always be superficial overlap at boundaries . If we had multiple groups, this will also help us seeing visually which groups are closer to each other relative to others .

We'll conclude our discussion with few not so obvious use cases of what we learnt today [apart from explicitly stated ones like consumer segmentation etc]

- You can use cluster membership as a feature in predictive modeling algorithm
- You can build different models for different segments
- You can use PCA to not only reduce dimensionality but at a conceptual level disentangle related information sets [variables].

Chapter 10 : Text Mining/ Text Analytics

In Text Mining, we use of automated methods in order to understand the content present in text documents. Its used to transform unstructured textual data to structured data useful for further analysis. It identifies and extracts information which would otherwise remain buried in the text.

Machine Learning usually requires well structured input features to work well, which is not the case when the data under consideration is textual. However, text mining can extract the clean, structured data from the text which is a requirement for most machine learning algorithms.

Some of the applications of text mining are as follows:

1.) Text categorization: e.g. categorizing emails into spam and not spam

2.) Text clustering: which can be to used to group related documents

3.) Sentiment analysis: to identify whether information in a document is positive, negative or neutral. e.g. Companies monitor social media to detect and handle any negative comments on their products or services

4.) POS or parts-of-speech tagging for a language: Each word in a document is tagged with its part of speech like noun, verb, pronoun, etc. This information can be useful for further analysis.

We have just touched the tip of the iceberg; there are many more applications using text mining.

We will be working with the NLTK package. NLTK is a Python package that is free, opensource and easy to use.

It assists us in doing the following tasks: tokenizing, part-of-speech tagging, stemming, sentiment analysis, topic segmentation, and named entity recognition among others. We will be discussing some of these tasks here.

NLTK, mainly helps to preprocess written text and converts it in a format that can be fed to the machine learning algorithms.

```
import nltk
import os
import pandas as pd
```

We can download NLTK data i.e the corpora (corpora is a large and structured set of texts) and other data from NLTK as follows:

```
nltk.download()
```

This download needs to be done only once.

For this discussion we will use Reuters data. We will understand how to gather data from multiple files. This data can be downloaded from https://www.dropbox.com/sh/865xeu4xwbyo2yt/AACd0pQEbeOSfvNjiV7aur4Ka?dl=0

On observing the data, the first thing we notice that the reuters_data folder consists of separate text files and each file contains a news article. We also observe that there are two categories of files - crude and money. On opening any file we read its news article.

In order to start processing the data we need to first access the path where all these individual text files are present:

```
path=r"/Users/anjal/Dropbox/0.0 Data/reuters_data/"
files=os.listdir(path)
```

Note: all the text data which we wish to analyze is not present in a single file.

If we wish to read the contents of a particular file, we can run the following code:

```
f=open(path+'training_money-fx_3593.txt','r',encoding='latin-1')
for line in f:
    print(line)
f.close()
```

> HUNGARY HOPES DEVALUATION WILL END TRADE DEFICIT
>
> National Bank of Hungary first
>
> vice-president Janos Fekete said he hoped a planned eight pct
>
> devaluation of the forint will spur exports and redress last
>
> year's severe trade deficit with the West.
>
> Fekete told Reuters in an interview Hungary must achieve at
>
> ⋮???
> ->Text Omitted<-

The code below goes through every line, ignores any empty line i.e. lines not containing any text or empty lines and keeps adding the lines containing text to the text variable.

```
f=open(path+'training_money-fx_3593.txt','r',encoding='latin-1') # handle to open the file

text="" # initialize variable to hold all the text for that file
for line in f: # read one line of the file at a time
    if line.strip()=="":continue # ignores empty lines
    else:
        text+=' '+line.strip() # adds lines with text to the variable text
print(text) # prints all the article text

f.close() # close the file

# Note - it is important to close the file after reading it. If we run the for loop again without closing the file, we will not
# get any content since after parsing the file the cursor is present on the last line of the file
```

> HUNGARY HOPES DEVALUATION WILL END TRADE DEFICIT National Bank of Hungary first vice-president Janos Fekete said he hoped a planned eight pct devaluation of the forint will spur exports and redress last year's severe trade deficit with the West. Fekete told Reuters in an interview Hungary must achieve at least equilibrium on its hard currency trade. "It is useful to have a devaluation," he said. "There is now a real push to our exports and a bit of a curb to our imports." The official news agency MTI said today Hungary would devalue by eight pct and it expected the new rates to be announced later today.
>
> ⋮
>
> ->Text Omitted <-

What we just discussed was for a single file.

Now, lets consider all the files present in the 'reuters_data' folder. We want to read from all the files present in this folder.

```
files[:10]
```

```
['.ipynb_checkpoints',
 '.RData',
 '.Rhistory',
 'training_crude_10011.txt',
 'training_crude_10078.txt',
 'training_crude_10080.txt',
 'training_crude_10106.txt',
 'training_crude_10168.txt',
 'training_crude_10190.txt',
 'training_crude_10192.txt']
```

On scrolling through the files, we observe some non text files i.e files without the .txt extension. We do not wish to read from the non text files. Hence, in the code below, we ignore the files which do not have a .txt extension.

```
target=[] # list containing the target variables
article_text=[] # list containing all the text present in all the files
for file in files: # read each file present in the folder at a time
    if '.txt' not in file:continue # Check if the file has a .txt extension; if not, then ignore the file
    f=open(path+file,encoding='latin-1') # open the file
    article_text.append(" ".join([line.strip() for line in f if line.strip()!=""])) #remove empty lines and join the text in

 # article_text list
    if "crude" in file: # check if the word crude is present in file name
        target.append("crude") # for the current file, append the target variable "crude"
    else:
        target.append("money") # for the current file, append the target variable "money"
    f.close() # we close the file each time we open it
```

In the code above we read each file present in the reuters_data folder, ignored the files without the .txt extension, removed the empty lines from the text and appended all the text in the 'article_text' list. For each file we also assign the target variable 'crude' or 'money'.

We then create a dataframe with the two lists created: target and article_text.

```
mydata=pd.DataFrame({'target':target,'article_text':article_text})
```

```
mydata.head()
```

	target	article_text
0	crude	CANADA OIL EXPORTS RISE 20 PCT IN 1986 Canadia...
1	crude	BP <BP> DOES NOT PLAN TO HIKE STANDARD <...
2	crude	BP<BP> OFFER RAISES EXPECTATIONS FOR OIL VA...
3	crude	USX <X> SAYS TALKS ENDED WITH BRITISH PETRO...
4	crude	BP <BP> MAY HAVE TO RAISE BID - ANALYSTS B...

We started with about 927 separate files; now the entire text data is present in a single dataframe having 927 rows.

```
mydata.shape
```

```
(927, 2)
```

Once the data is converted to a single dataframe from individual files, we can now start exploring its content.

Exploring text data with wordcloud

Word cloud is an image made of words only used in a given text, in which each words size indicates its frequency or importance.

Word clouds help in identifying patterns in the text that would be difficult to observe in a tabular format. Higher the frequency of a word in the text, the more prominently it is displayed.

The code to visualize wordcloud of text data is as follows:

```
from wordcloud import WordCloud
```

```
import matplotlib.pyplot as plt
%matplotlib inline
```

We first need to combine the text into bunches for which we want to see the wordclouds.

```
all_articles=' '.join(mydata['article_text']) # to create wordcloud for all the articles; we join the entire column into a
                                              # single text value
```

```
all_articles[:1000]
```

> 'CANADA OIL EXPORTS RISE 20 PCT IN 1986 Canadian oil exports rose 20 pct in 1986 over the previous year to 33.96 mln cubic meters, while oil imports soared 25.2 pct to 20.58 mln cubic meters, Statistics Canada said. Production, meanwhile, was unchanged from the previous year at 91.09 mln cubic feet. Natural gas exports plunged 19.4 pct to 21.09 billion cubic meters, while Canadian sales slipped 4.1 pct to 48.09 billion cubic meters. The federal agency said that in December oil production fell 4.0 pct to 7.73 mln cubic meters, while exports rose 5.2 pct to 2.84 mln cubic meters and imports rose 12.3 pct to 2.1 mln cubic meters. Natural gas exports fell 16.3 pct in the month 2.51 billion cubic meters and Canadian sales eased 10.2 pct to 5.25 billion cubic meters. BP <BP> DOES NOT PLAN TO HIKE STANDARD <SRD> BID British Petroleum Co Plc does not intend to raise the price of its planned 70 dlr per share offer for the publicly held 45 pct of Standard Oil Co, BP Managing Director David '

```
crude_articles=" ".join(mydata.loc[mydata['target']=='crude','article_text']) # to create the wordcloud for crude articles
```

```
money_articles=" ".join(mydata.loc[mydata['target']=='money','article_text']) # to create the wordcloud for money articles
```

The wordcloud can be generated by passing a text variable (i.e. all_articles, crude_articles or money_articles) to the WordCloud() function as shown below.

Below is the wordcloud made by considering all the articles.

```
wordcloud = WordCloud().generate(all_articles)
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis("off")
```