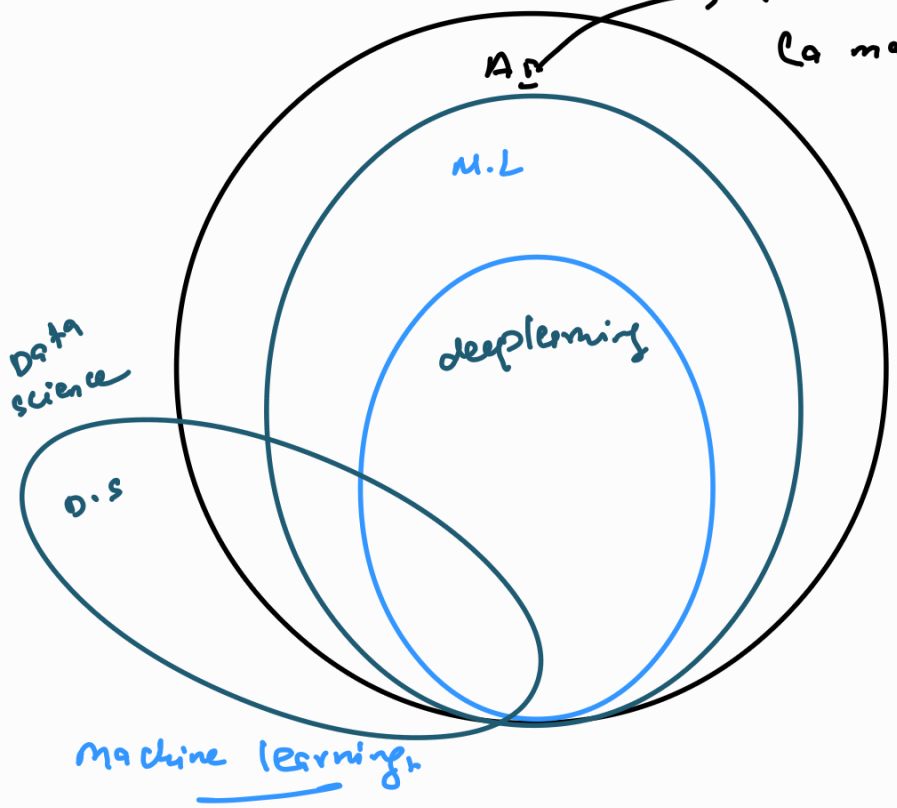
What is machine learning?

Machine learning is when computers learn from data to make decisions or predictions without being told exactly what to do.

input → the data we give

Learning → happens when computer finds out a pattern in that data

prediction: when the computer uses what it learned to make a guess about new data.



human intelligence exhibited by machines (a machine that acts like human)

Narrow AI → machines that can be as good or even better than humans at specific tasks.

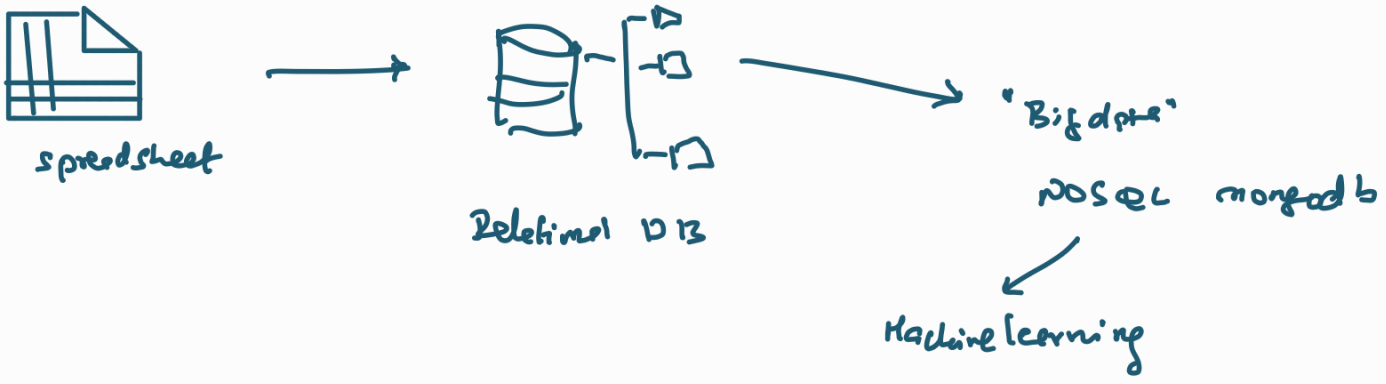
General AI: with multiple abilities like humans, something we are very far away from now.

Stanford describes machine learning as the science of getting computes to act without being explicitly programmed.

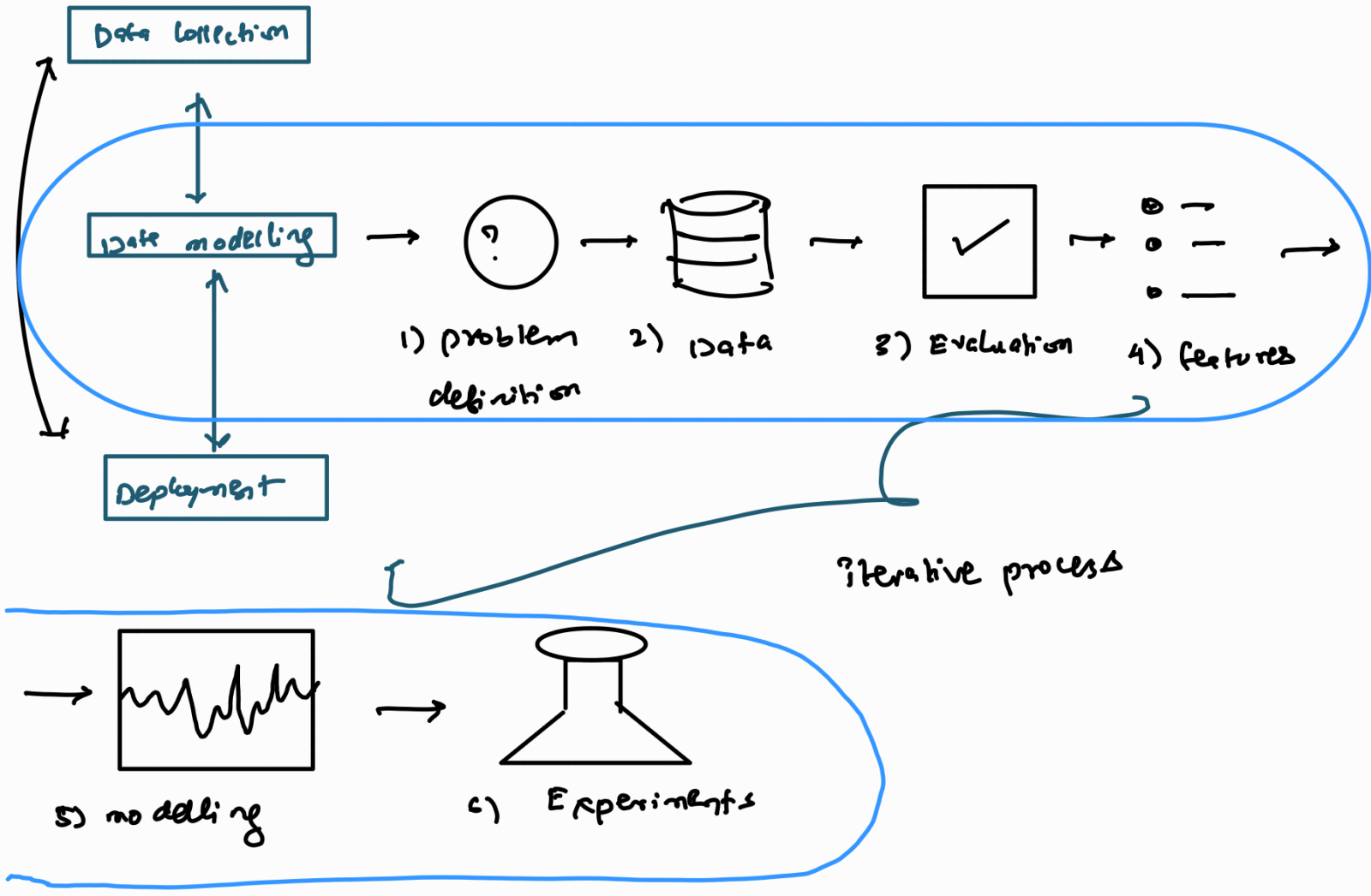
Deep learning / deep neural network:

It is one of the techniques for implementing machine learning. (Something like an algorithm)

Why AI/ML?



Steps



1) What problem we are trying to solve?

2) What data do we have?

3) What defines success?

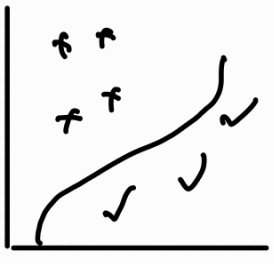
4) What features should we model?

5) What kind of model should we use?

6) What we have tried / what else can we try?

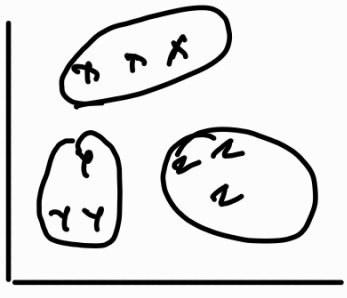
Machine learning

Supervised



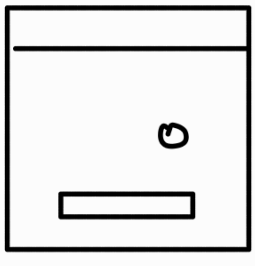
classification

unsupervised



Clustering

reinforcement

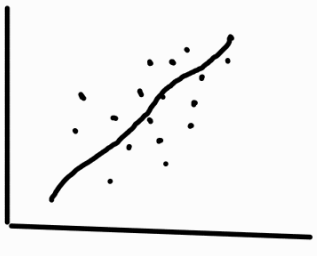


Skill acquisition

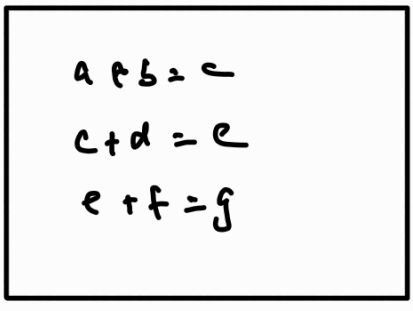
Real time learning

[trial & error method]

based on rewards & punishment



regression



Association rule learning

Machine learning is all about predicting results based on incoming data and all these categories simply do that.

The framework.

i) Create a frame work
ii) Match to data science and machine learning tools
iii) Learn by doing

6 Step machine learning framework

i) problem definition
ii) Data
iii) Evaluation
iv) features
v) modelling
vi) Experiments

i) problem definition:

What type of problem are we solving

i) supervised
ii) unsupervised
iii) classification
iv) regression

2) Data.

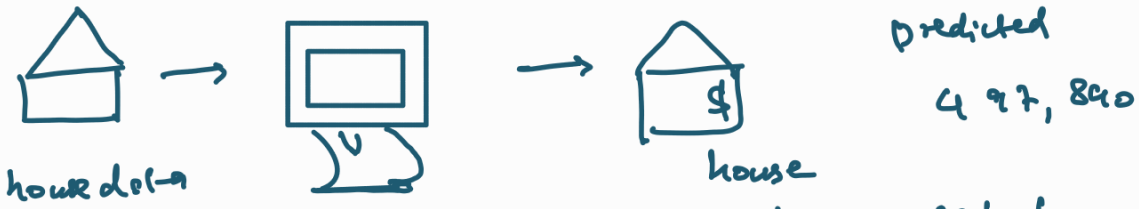
What kind of data do we have

i) structured data like excel, Rdbms

ii) unstructured data like nosql, mongodb

3) Evaluation

What defines success to us?



like 95?

4) features

What do we already know about data?

The work of ML is to evaluate these features and make predictions

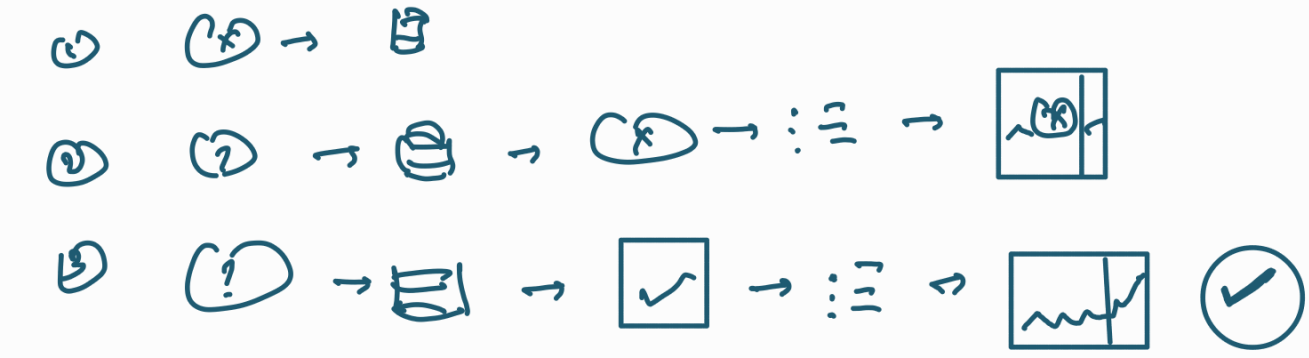
5) Modelling.

Based upon our problem and data, what model should we use?

6) Experimentation.

How could we improve / what can we try next?

Attempt



1) When to not use machine learning?

When a simple code works.

Main types of machine learning

i) supervised learning
ii) unsupervised learning
iii) transfered learning
iv) Reinforcement learning

S.L

Classification

Is this example one thing or the other?

Binary classification - two options

Multiclass classification - more than two options.

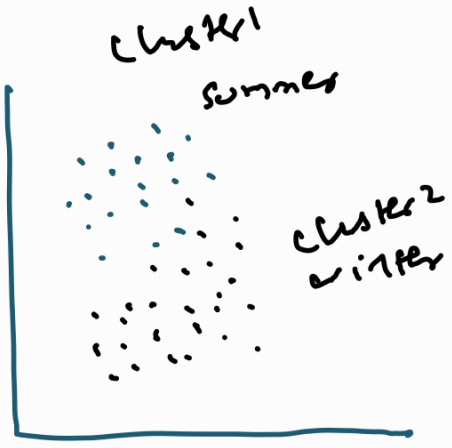
Regression ?

How much will this house sell for?

How many people will buy this app?

VL.

id	Par1	Par2
1	Sung	a
2	Jac	b
3	Sons	c

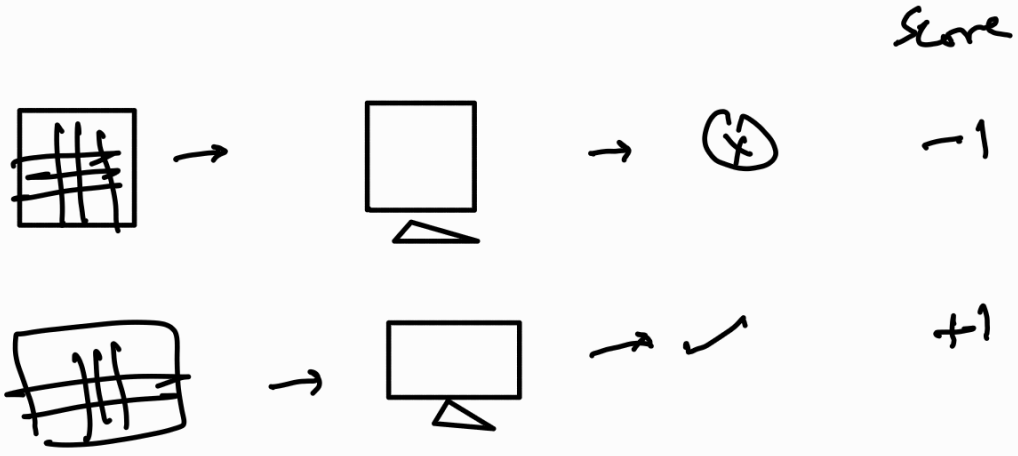


Transfer learning.

We can fine tune already existing algorithm(models) to predict other things.

like we can use a CIF model & use it for dog model.

Reinforcement learning.



It involves having a computer program perform some actions within a defined space.

A good example is teaching a machine learning algorithm to play chess

2.

Types of data

i) structured

ii) unstructured

iii) streaming data - constantly changing data like stocks

Types of evaluation

what is success for us?

different types of metrics

Classification	regression	recommendation
		precision at k
Accuracy	mean abs error (MAE)	
precision	mean squared error (MSE)	
Recall	root mean squared error (RMSE)	

features in data

what do we already know about the data?

feature variables

target variables

id	weight	gender	heart rate	chest pain	HD
1	60	M	81	4	Y
2	70	F	82	1	N
3	80	M	65	2	N
4	90	F	77	0	Y

Numerical features
Categorical features
derived features
} feature Engineering.

feature coverage : that enough relevant data is present (like more than 10%.)

ideally 100%.

3

modelling data :- Splitting data

training validation testing — 3 sets

1) Splitting data
2) picking the model
3) tuning the model.

↓ split

3 sets

train model on this 70-80%

tune model on this 10-15%.

test & compare on this 10-15%.

should not memorize, instead learn, so train, validate & test

all 3 must be different

Generalisation:

The ability for a machine learning model to perform well on data it hasn't seen before

(2) Picking the model : Choosing a model & training data

Structured data ⟶ catboost, XGboost, random forest

unstructured data ⟶ deeplearning, transfer learning, neural network

Then train the model, after choosing one

input → output x

input → output ✓

input → output x

input → output ✓

Goal : minimise time between experiments

Ex 1 :

		Acc	time
input → model 1 → output		87	1m
input → model 2 → output		91	80m
input → model 3 → output		94	90m
input → model 4 → output		95	120m

Things to remember

→ Some models work better than others on different problems

→ Don't be afraid to try things

→ Start small & build up (add complexity as needed.

Up next?

1) Choosing and training a model
   ↳ training data

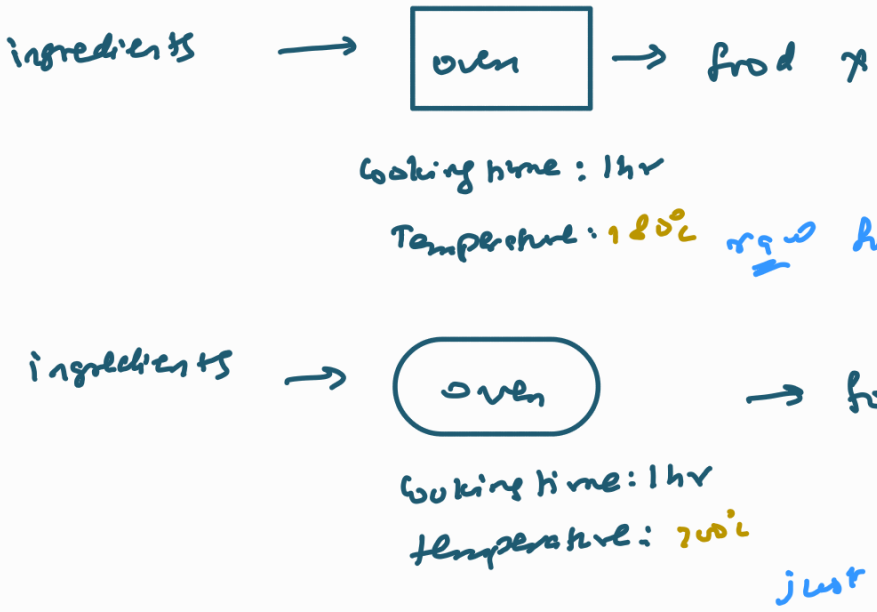
2) Tuning a model
   ↳ validation data

3) Model comparison
   ↳ Test data

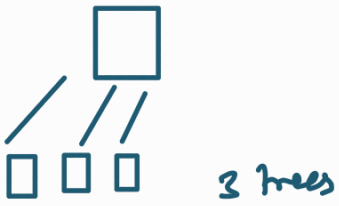
Modelling - Tuning

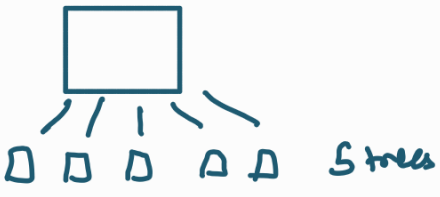
9

Tuning a model

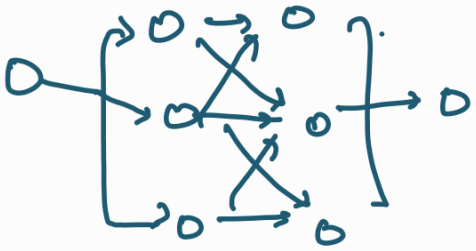


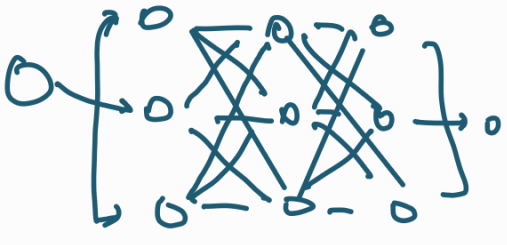
Random forest





neural network





→ machine learning models have hyperparameters that we can adjust
→ A models first result aren't it's best
→ Tuning can take place on training or validation data sets.

Modelling

Comparison

this happens during experimentation.

how will our model perform in real world?

After we have churned and improved our model's performance through hyper parameter tuning, it's time to see how it performs on test set.

A test set is like a final exam for machine learning models.

if we have created our data splits cructly, it should give an indication on how model will perform once deployed in the production (since it has not seen the test)

→ A good model will yield similar results on training, validation and test sets.

→ it not uncommon to see a decline in performance from the model on the training and validation set to the test set

Testing a model

✓

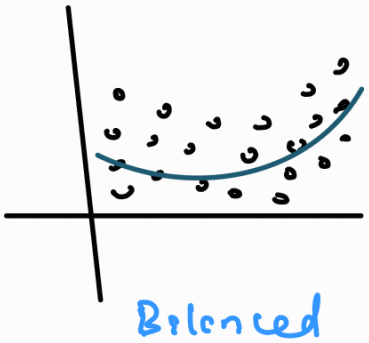
Data set	performance
Training	98 %
test	96 %

Data set	performance	
Training	64 %	underfitting potential
test	42 %	

Data set	performance	
Training	93 %	overfitting potential
test	99 %	

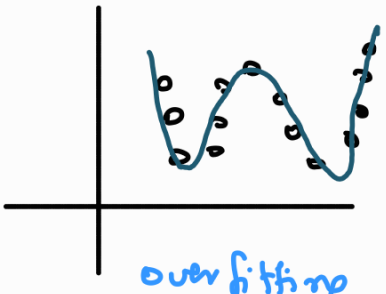
We do not want underfitting or overfitting.

Underfitting



Balanced

(goldilock zone)



overfitting

Underfitting and overfitting can happen due to several reasons like

- Leakage
- data mismatch

Data leakage happens when some of our test data leaks into the training data and this often results in overfitting or a model doing better on the test set than on the training set

ensure machine learning model training happens only on training data set validation and model tuning happens on validation or training data set testing and model comparison happens on test data set

Some approaches use only training and testing set and do model tuning on training set

Data leakage

Data mismatch:

Data mismatch happens when the data you're testing on is different to the data you're training on.

Having this kind of mismatch can lead to models performing poorly on test data compared to training data.

underfitting

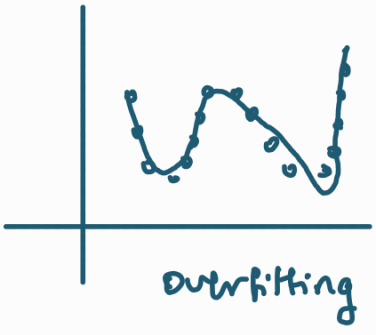
So: it is important that training is done on the same kind of data as the testing and as close as possible to what you'll be using in your future applications.

We can combat underfitting using

i) more advanced models
ii) increase model hyperparameters
iii) Train longer
iv) reduce the number of features.

To combat overfitting use:

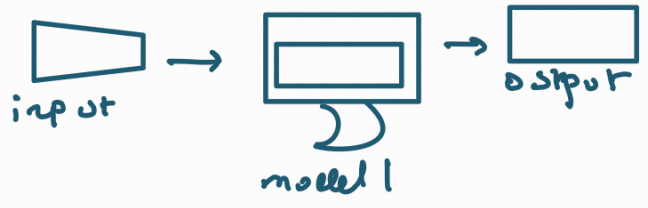
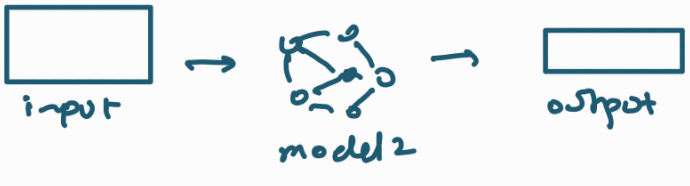
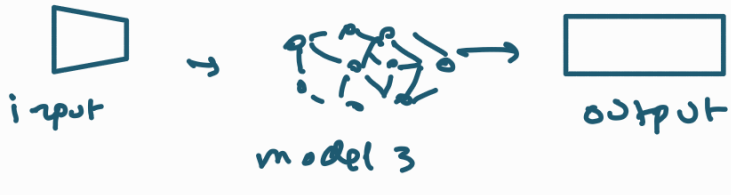
→ Collect more data
→ Try a less advance model



Comparing model.

[Compare models that had same inputs (ie) apple with apples]

Experiment

		Accuracy	training time	prediction time
1)		87.5%	3 min	0.5 sec
2)		91.3%	92 min	1 sec
3)		94.7%	176 min	4 sec

Things to remember:

i) avoid overfitting and underfitting (head towards Generality)

ii) keep the test separate at all costs

iii) Compare apples to apples (ie same data set)

iv) one best performance metric does not equal best model

Summary

All experiments should be conducted on different portions of data.

Training data set: use this set for model training, 70-80% of data is the standard.

validation data set: use this set for model hyperparameter tuning and experimentation evaluation, 10-15% of the data is the standard.

Test data set:

for model testing and comparison, 10-15% of the data is the standard

Poor performance:

means that the model hasn't learned properly and is underfitting.

→ try a different model, improve the existing one through hyperparameter tuning or collect more data

Great Performance:

Great performance on training data set but poor

performance in test data set (does not generalize well).

Model must be overfitting.

→ Use a simpler model

→ Collect more data.

Another form of overfitting could be that the model is performing better on test data than on training data set.
Then data leakage must have happened.

→ Poor performance once deployed means there's a difference in what you trained and tested the model on and what is actually happening.

Ensure the data you're using during experimentation matches up with the data you're using in production.

Step 6) Experimentation:

Steps

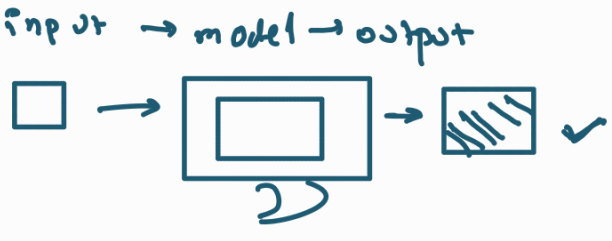
- Data collection
- Model (problem definition → data → Evaluation → features → modelling → Experiments)
- (problem)
- deployment

data analysis

machine learning model

can it be better??

iterative



Tools we Use :

your computer

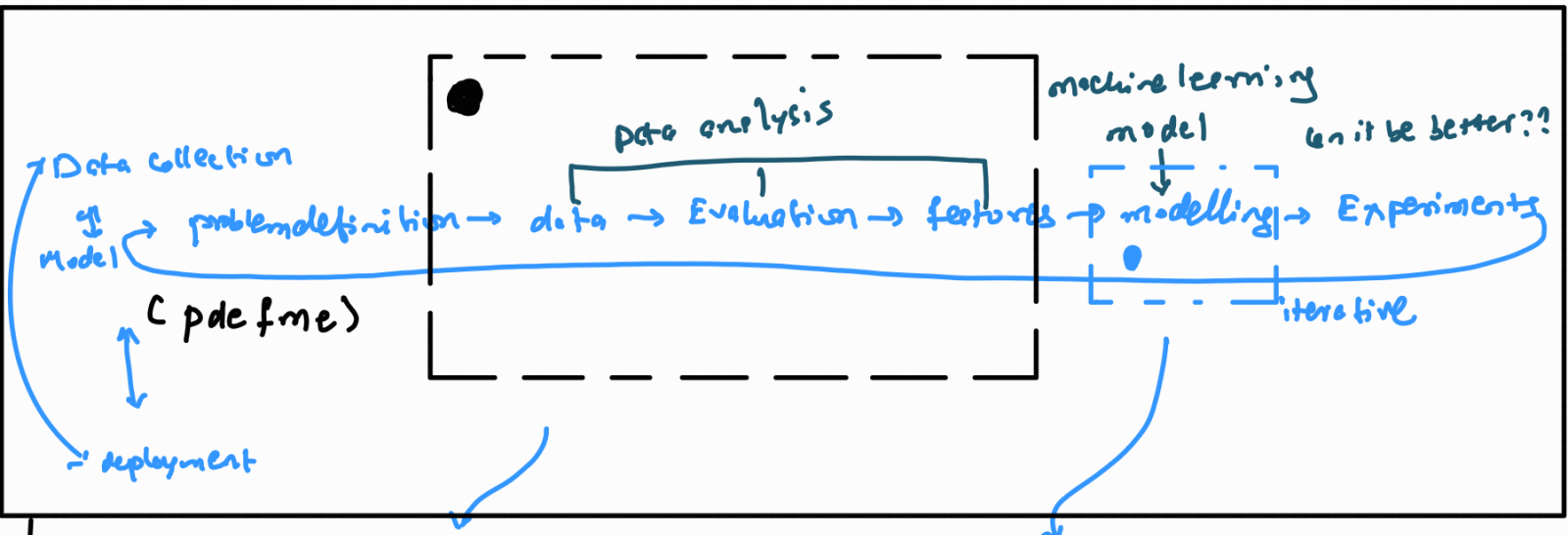
Anaconda

jupyter matplotlib seaborn

numpy pandas Tensorflow pytorch

scikit learn XGboost

mapping steps to what we will use . . .



Data analysis

pandas numpy matplotlib

modelling

Tensorflow pytorch XGboost

Catboost

jupyter note books

Anaconda

2 - Paths

knows python

does not know python.

Setting up environment.

Software distribution:
- i) install Anaconda (or) (like a store) → ~3.4 GB
- miniconda (or) (like a workbench) → ~200 MB

package manager ← conda (like an assistant) → comes along with miniconda
(to install/uninstall other packages)

---

use conda and install

↳ conda install pandas numpy matplotlib ctkern

python → [just enough to get you Hired]

(cpython vm)

first python code..

Create a file → hello.py

```
print("hello")
```

>> python hello.py

>> hello

```
name = input("enter name : ")
print(name)
```

>> enter name: jay

>> jay

Python Datatypes

fundamental datatypes

i) int eg 1, 2

ii) float eg : 1.2

iii) str eg "a"

iv) bool True, false

v) None

vi) list eg [1,2,3]

vii) tuple (same as list but immutable) eg (1,2,3)

viii) set

ix) dict

x) Complex (something like 1+2j) → j is imaginary

(optional) ↗

xi) bin()

2+2j

```
print((1+2j)+(2+2j))
>> (3+4j)
```

classes

Custom types

Specialized data types: Modules

---

int

for numbers

eg =

```
int1 = 10      (we can also use "_" as seperator 100000 → 100_000)
int2 = 20

print (int1 + int2)
```

float

for decimals

eg =

```
float1 = 1.23
float2 = 1.2345

print (float1 + float2)
```

math operations

+	add
-	sub
*	mul
/	div
%	mod
//	floor div
**	raise to power (exponent)

math functions

max	len	abs
min	round	sqrt
sum	ceil	
	floor	

str: (Strings, ordered)

for strings, strings are immutable

eg =

```
name = 'jay'

print(name)
print (name[0]) → j
```

formatted string.

f" "

```
>> print(f"name is {name}")
>> name is jay
```

Concat :

```
name = jay
location = ind
print( name + " " + location)
>> jay ind.
```

```
print(f"{name} {location}")
>> jay ind
```

Slicing : name [start : stop : step]

List.. [Just like arrays, ordered and heterogeneous]

eg

```
list1 = [ 1, 2, 3, 'a', 'b', 'c', True, false ]
print (list1)
print (list [0]) → 1
```

Tuple : [same as list but is immutable]

eg: Tuple1 = (1, 2, 3, 'a', 'b', 'c', True, false)

print (tuple1) → (1, 2, 3, 'a', 'b', 'c', True, false)

print(Tuple [0]) → 1

Set: kind of list but is unordered, no duplicate values,

eg

```
set1 = set()  or  set = {1, 2, 3}
set.add(1)
set.add(2)
set.add(2)
print (set1)
```

(#) print(set[0]) → error, as sets don't support indexing

Dict: similar to map ie with key value pairs.

eg

```
student = { 'name' : 'jay', 'age' : 25, 'location' : 'india' }
print( student)
```

→ { 'name' : 'jay'
'age' : 25,
'location' : india }

```
print ( student['name'])
```

↳ jay

None: null or nothing data type.

we cannot declare the variables alone; so we initialize with None.

bool: represents True or false states.

Developer fundamentals:

i) Don't read the Dictionary

(ie) Don't try to learn everything

Math precedence

PEMDAS - L R

Parenthesis

Exponent

Multiply / Divide (from left to right)

Add / Sub (from left to right)

Bin() data type.

```
bin(5)
>> 0b101

print(int('0b101', 2))   # 2 → binary
>> 5
```

Variables:

→ a container to store values.

eg: a = 5
- a → variable
- 5 → value

a = 5 → Binding

a, b, c = 1, 2, 3

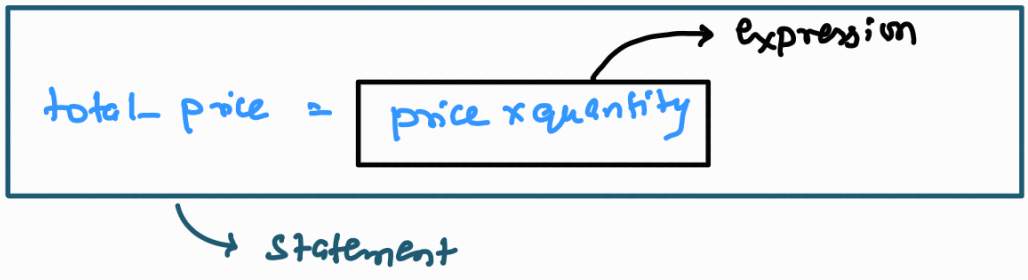
best practices

i) snake_case
ii) start with lowercase or underscore
iii) letters, numbers, underscores
iv) case sensitive
v) Don't overwrite keywords

- _name → protected variable
- __name → private variable
- name → public variable

Expressions vs Statements.

eg:



Augmented Assignment Operator. (Short hands)

some_value = 5

some_value = some_value + 2 → some_value += 2

Other Augmented operators

+=
-=
*=
/=
//=
**=
>>=
<<=

Type Conversion:

why to type convert?

→ We cannot string concat different datatypes, so we need to convert it to required datatype

→ Api requests are sent in the form of strings, we may need to extract values that are not string.

→ or lets say input() , it takes string by default, we have to type cast it to int or float if input is required in number format

eg.

```
a = input("num1")
b = input("num2")
print(a+b)
```

```
>> num1 1
   num2 2
>> 12
```

issue

type conversion

```
a = int(input("num1"))
b = int(input("num2"))
print(a+b)
```

```
>> num1 1
   num2 2
3
```

Escape sequence

"\"

to represent special characters (like newline \n, tab \t, quotes \", \') within strings without breaking the syntax.

eg:

str1 = 'it's sunny' → error

↓

'it\'s sunny'

str = "it's "kind of" sunny"

↓

"it's \"kind of\" sunny"

Built-in functions.	Built-in methods
len()	.upper()
range()	.lower()
max()	.strip()
min()	.isdigit()
sum()	.isalnum()
str()	.isalpha()
int()	.split()
	.capitalize()
	.find("word")
	.replace(replace, with)

Developer fundamentals ii:

i) add comments to code. (#comment)

ii) do not add comments to self explainatory

List : a collection of items.

```
amazon_cart = [ "notebooks", "sunglasses"]
print (amazon_cart [1])
→ sunglasses
```

List slicing (just like string slicing, but are mutable)

```
amazon_cart = [ "notebook",
                "sunglasses",
                "toys",
                "groples" ]
```

```
→ print [ : :2]
→ ['notebook', 'toys']
```

Copy list

1) `amazon_cart1 = amazon_cart`

This creates a shallow copy (i.e.) change in one reflects in another

2) `amazon_cart1 = amazon_cart [:]`

Changing one list does not affect other

Matrix → 2D list

```
matrix = [[],[]]
```

eg

```
matrix = [
        [1,2,3],
        [4,5,6]
        ]

print(matrix[1])
>> [4,5,6]

print(matrix[0][1])
>> 2
```

List methods: basket = [1,2,3]

i) .append()
   ↳ appends in place.

ii) .insert(index, element) → in place

iii) .extend
   ↳ can be used to append a list to existing list (in place)

iv) .pop() (pop returns what's popped)
   ↳ removes the last element

v) .pop(index)
   ↳ removes at indx

vi) .remove(value) (does not return)
   ↳ removes the value

vii) del list[index] / [start:end] } removes at index

viii) clear() → removes everything

ix) .index(value, [start, stop]) (throws error if not found)
↳ optional

x) in eg: print(1 in basket)
>> true

xi) .count('value')

xii) .sort (does in place sorting)

xiii) .sorted (creates a copy, original is unchanged)

xiv) .copy()

xv) .reverse() (in place)

Common list patterns:

list1 = [1,2,3]

list.reverse() | list[::-1]

does inplace | creates a new list

print(list(range(100)) → [0,1,2...99]

.join()

mostly used like: "".join(list)

eg = list3 = ["my", "name", "is", "jay"]

print(" ".join(list3))

my name is jay

List unpacking

①
```
a, b = [1, 2]
print(a, b)
```
>> 1 2

② List assignment
```
[a, b] = [1, 2]
print(a, b)
```
1 2

both are pretty much same ① is preferred tho

```
a, b, c, *d, e = [1, 2, 3, 4, 5, 6, 7]
```
print(a) → 1
print(b) → 2
print(c) → 3
print(d) → [4, 5, 6]
print(e) → 7

None = null

eg
```
game
weapon = None
print(weapon)
```
>> None

Dictionary (dict) → map, hashtable [unordered]
↳ datatype and data structure

```
dictionary = {
        'a' : 1
        'b' : 2 }
```

```
print(dictionary['a'])
>> 1
```

Developer fundamentals : iii

i) know which data structure to use where.

ii)

Dictionary keys:

keys should be hashable type (ie) constant/immutable

(ie) we can use tuple as key but not list.

Dictionary methods:

1) .get() → avoids error if a property does not exist

2) user2 = dict('name' : 'jay') (creates a dict)

3) in

4) .keys()

5) .values()

6) .items() (both key, value)

7) .clear()

8) .copy()

9) copy.deepcopy(original obj)

10) .pop()

11) .popitem() → pops randomly

12) .update({'key' : value})

Tuple

immutable lists

```
my_tuple = (1,2,3)

tuple1 = (1,)
```

methods

- .count(value)
- .index(value)
- .len

Sets

unordered collection of unique object.

eg

```
my_set = {1,2,3,4,5,5}

print(my_set)
```

>> {1,2,3,4,5}

methods

- .add()
- .clear()
- .pop()
- .remove()
- .discard()
- .copy()
- .in
- .difference()
- .union() → '|'
- .intersection() → '&'
- .difference_update() → modifies in place

remove duplicates from list

```
my_list = [1,2,3,4,5,5,6,6]

print(list(set(my_list)))
```

>> [1,2,3,4,5,6]

isdisjoint( ) → check if nothing is common

.issubset( )

.issuperset( )

Conditional logic.

→ if

→ if else

→ if elif else

i) if condition :

    task

ii) if condition :

    task1

  else :

    task 2

iii) if condition :

    task 1

  elif condition :

    task 2

  else :

    task 3

Indentation.

Python does not use {} braces for code blocks, so it uses identation instead

True and false values.

→ Anything apart from falsy is true

falsy values

None
False
0
0.0
0j
Decimal(0)
Fraction(0,1)
[] empty list
{} empty dict
() empty tuple
'' empty str
range(0)

Ternary Operators.

Conditional expression.

Syntax

condition-if_true if condition else condition-if-else

Eg: is_friend = False

can_message = 'message allowed' if is_friend else "not allowed"

print(can_message)

>> message allowed

Short circuiting.

Eg

is_friend = True
is_user = False

```
if is_friend and is_user:
    print("friends")
```
>> ____

```
if is_friend or is_user:
    print("friends")
```
>> friends

logical operators (returns true or false)

and
or
not
>
<
==
>=
<=

```
print('a' < 'A')
```
>> false

---

is vs ==

print(True == 1) → True

print('' == 1) → false

print([] == 1) → false

print(10 == 10.0) → True

print([] == []) → True

print('1' == 1) → false

== → checks only for values (converts one data type to another)

is → checks if value and data type are same, also checks for location

print(True is 1) → false

print('' is 1) → false

print([] is 1) → false

print(10 is 10.0) → false

print([] is []) → false

Loops:

- for → when number of iteration is known
- while → when number of iterations is not known

for:

- for item in 'something':
  task
  ('something' → iterable)

eg
```
for item in 'something':
    print(item)
```
→
```
s
o
m
e
t
h
i
n
g
```

- using range (indexes)

```
for i in range(start, stop, step):
    action
```

eg
```
for i in range(0, 5, 2):
    print(i)
```
→
```
0
2
4
```

iii) using enumerate. (index, num)

```
list1 = [1,2,3,4]

for index,num in enumerate(list1):
    print(num, count)
```

```
>>  0 1
    1 2
    2 3
    3 4
```

---

iterable:

↳ a collection of items that can be iterated.

eg: list, tuple, dict, str, set

```
dic = {
    "name" : "golem",
    "game" : "col",
    "can-swim" : True
    }
```

```
for item in dic:
    print(item)
```

```
>>  name
    game
    can-swim
```

```
for key, value in dic.items():
    print(key, value)
```

```
>>  name    golem
    game    col
    can_swim    True
```

```
for key in dic.keys():
    Print(key)
```

```
>>  name
    game
    can-swim
```

```
for value in dic.values():
    print(value)
```

```
>>  golem
    col
    True
```

Count items using loop.

eg: list1 = [1,2,3,4]

```
count = 0
for item in list1:
    count += 1
print(count)
```

>> 4

```
count = 0
for _ in range(len(list1)):
    count += 1
print(count)
```

>> 4

(_) → is variable is not needed

Simpler way to count?

```
print(len(list1))
```

>> 4

Range (start, stop, step)

- step → increment/decrement
- stop → last value - 1 (i.e last value is not included)
- start → start (included)

Enumerate:

Syntax: enumerate (iterable)

eg:

```
list1 = [1,2,3,4]
for ind, num in enumerate(list1):
    print(ind, num)
```

```
>>  0 1
    1 2
    2 3
    3 4
```

```
enumerate(list(range(10)))
```

while

Syntax : while condition:
    task

```
while condition:
....
else:
....
```

```
count = 0
while count < 5:
    print(count)
    count += 1
```

→ Since 0 < 5 always, this would cause infinite loop. (so have a base condition or break)

```
>>
0
1
2
3
4
```

Break, continue, pass

break:
    to break out of loops.

continue:
    goes back to the loop

pass:
    ↳ does nothing (mostly used as placeholder (...))

Developer fundamentals iv.

i) clean, good code, readability,
ii) Predictability
iii) DRY - do not repeat yourself

functions:

Syntax

```
def funname(param):
    code

funname(args) → function call
```

eg

```
def greet(name):
    print(f'hello {name}')

greet('jay')
```

>> hello jay

why functions?

i) to create our own functions

ii) to keep code modular

iii) to avoid code repetition.

Parameters and arguments.

parameters (in function)

```
def funname(parameters):
    code
```

arguments → in calling function

```
funname(arguments)
```

positional arguments.

(arguments that require to be in proper position)

keyword arguments (not positional) → like dict

eg:- def say_hello(name, emoji) :

print(f" hello {name} {emoji} ")

keyword argument

say_hello(emoji = '🐻', name = 'bibi')

>> hello 🐻 bibi

default parameters.

eg:- def say_hello (name = "user", emoji = '-_-'):

printC f" hello {name} {emoji}")

→ say hello ()

>> hello user -_-

Return :

→ multiple parameters

def add_name (*args):

return sum(args)

print (add_num (1,7, 3,4))

>> 10

→ A function should return something

A return keyword exits the function.

Methods vs functions.

↓

.len()

.count()

.upper()

.lower()

.capitalize()

↳ list()

print()

max()

min()

input

user-defined-functions()

Doc Strings. ⟶ (allows to comment inside the fun)

Syntax: """ """ or ''' '''

eg. def test(a):

"""

Info: this function test and prints parameter

"""

print(a)

test('!!!')

>> !!!

help(test)

>> Info: this function test and prints parameter

print(test.__doc__)

>> Info: this function test and prints parameter

clean code.

eg:- isOddEven

```
num = int(input("Enter num:"))
print("odd" if num%2!=0 else "even")
```

```
>> Enter num:4
   even
```

```
def isOddEven(num):
    return ("odd" if num%2!=0 else "even")

print(isOddEven(4))
```

```
>> even
```

---

*Args , **kwargs

*args

```
def super_fun(*args):
        sum(args)

super_fun(1,2,3,4)
```

```
>> 10
```

**kwargs

```
def super_fun(*args, **kwargs)
        print(kwargs)
        return sum(args) + sum(value for value in kwargs.values())

print super_fun(1,2,3,4, num1=5, num2=5 )
```

args ↓ (1,2,3,4) — kwargs → (num1=5, num2=5)

```
>> 20
```

Rule: params , *args , default params , **kwargs

Scope:

↳ what variables do I have access to

i) local scope (inside functions, has a parent?)

ii) Global scope (has no parent)

Global → no local, no parent, no indentation.

→ built in python function

Global keyword.

eg

```
total = 0

def count():
    global total
    total += 1
    return total
count()
count()

print(count())
```

>> 3

to access outer global value inside local function.

Better way? yes, dependency injection. (passing as a param)

```
total = 0

def count(total)
    total += 1
    return total

print(count(count(count(total))))
```

>> 3

Non local: (closures)

```
def outer():
    x = "local"
    def inner():
        nonlocal x
        x = "nonlocal"
        print("inner", x)
    inner()
    print("outer", x)

outer()
```

=> nonlocal
nonlocal

Pure functions.

→ Given the same input, it will return same output
→ A function should not produce side effects.

```
def multby2(li):
    newlist = []
    for item in li:
        newlist.append(item*2)
    return new_list

print(multiplyby2([1,2,3])
```

→ it is more like a guideline (makes debugging easy)

Map(), filter, zip(), reduce()

i) Map()

Syntax: mapl fun , *iterable)

eg. my_list = [1,2,3,4]

new_list = list(map(lambda value : value * 2, my_list))

print (new_list)

>> [2,4,6,8]

ii) filter()

even = list(filter (lambda value : value % 2==0, my_list))

print (even)

function — iterable

>> [2,4]

iii) zip()

list1 = [1,2,3]

list2 = [4,5,6]

print (list(zip(list1, list2)

>> [(1,4), (2,5), (3,6)]

iv) reduce()

from functools import reduce

Syntax: reduce (function, iterable [, initializer])

eg. my_list = [1,2,3,4]

result = reduce (lambda x,y : x+y , my_list, 0]

print (result)

accumulator

>> 10

Comprehensions: quick way to create lists, sets, dictionaries.

→ list, set, dictionaries.

List comprehensions

```
name = "golem"
my_list = [ char for char in name]
  print(my-list)
>> ['g', 'o', 'l', 'e', 'm']
```

```
→ nums = [num for num in range(0, 101) [if condition]]
   print (nums)
>> [0, 1, 2 ..... 100]
```

(expression)

Set comprehensions

Same as list but { } instead of [ ]

```
name = "golem"
my_set = { char for char in name}
  print(my-set)
>> {'g', 'o', 'l', 'e', 'm'}
```

```
→ nums = { num for num in range(0, 101) [if condition] }
   print (nums)
>> {0, 1, 2 ..... 100}
```

(expression)

Dictionary comprehension

```
simpledict = {
        'a' : 1 ,
        'b' : 2
    }
my_dict = { key: value *2  for key, value in simple dict.items() }

print (my_dict)
```

>> { 'a' : 1 , 'b' : 4 }

→ to use list values as keys

```
my_dict = { nums : nums* 2 for num in [1,2,3] }
print (my_dict)
```

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

---

find duplicates in a list using list comprehension.

```
my_list = [ 1, 2, 3, 4, 5, 5, 6, 6, 7, 7, 8]
duplicate = [value for value, count in (counter( my_list ).items() )
                                        if count > 1 ]
Print (duplicate)
```

→ [5, 6, 7]

Modules: A module is a single .py file

A way to organize files

lets say 2 files

i) main.py

ii) utility.py

in main.py we can use utility.py by "import utility"

```
>> print(utility.divide(4,2))
   >> 2.0
```

Packages:- Package is a directory that contains modules.
(it has a __init__.py) → tells python that it is a package.

folder
- module1
- module2

To import from folder

```
import folder.module
```

Different ways to import:

i) from packagename import module.

ii) import whole module
eg → import math

iii) import with alias
eg → import numpy as np

iv) from specific objects
→ from math import sqrt

v) dynamic import using
__import()__

eg
```
mod = __import__('math')
print(mod.sqrt(49))
```

vi) relative import

```
from . import siblingModule
from .. ParentPackage import
        something
```