(*) Stemming and Lemmatization :-

→ Process of reducing infected words to their word STEM.

Stemming

history, historical → histori

finally, final, finalized → fina

Lemmatization

history, historical → history

finally, final, finalized → final

gone, goes, going → go

NOTE::

(1) Lemmatization the converted words will have meaning whereas in stemming in some of the cases it is not possible.

(2) Lemmatization takes more time than stemming.

Q. where we can use stemming and lemmatization?

→ Stemming :- Sentiment Analysis, Spam classifier.

Reason :- In this cases we require only base word. we are having the words in our dataset.

Lemmatization :- chatBot, Question Answer App

Reason :- The response that human gets from app should be meaningful.

NOTE:: (1) Lemmatization is slower and accurate.

(2) Stemming is faster but inaccurate.

– Practical implementation of stemming :-

```
import nltk
from nltk.stem import PorterStemmer
from nltk.corpus import stopwords
paragraph = """ ____ """
sentences = nltk.sent_tokenize(paragraph)

stemmer = PorterStemmer()
# Now we will remove stop words from each sentences...
# Then after removing the stop words we will stemming
# each word using stemmer.
for i in range(len(sentences)):
    # Convert each sentence into words...
    words = nltk.word_tokenize(sentences[i])

    # Stemming the words...
    words = [stemmer.stem(word) for word in words
             if word not in
             set(stopwords.words
             ('english'))]

    #Join the words
    sentences[i] = ' '.join(words)
```

for stemming

e.g:- of, them, from the, etc... this words do not play an imprtnt role in +ve/-ve sentiment analysis.

convert paragraph into sentences

converting sentences into words

stemming the words which are not stopwords.

Problem with stemming :-

→ Produced intermediate representation of the word may not have any meaning.

Example:- fina, histori, etc.

(*) Practical implementation of lemmatization :-

responsible for lemmatization

```
import nltk
from nltk.stem import WordNetLematizer
from nltk.corpus import stopwords
paragraph = "  "

#convert para into sentences...
sentences = nltk.sent_tokenize(paragraph)

#creating object of WordNetLematizer...
lemmatizer = WordNetLemmatizer()

#Lemmatizing...
'''
1. creating words from sentences.
2. lemmatizing words which are not stopwords.
3. joining the words to form the sentences.
'''
for i in range(len(sentences)):
    #1
    words = nltk.word_tokenize(sentences[i])
    words = [lemmatizer.lemmatize(word) for word in words
             if word not in
             set(stopwords.words
             ('english'))]

    #3. Join the words...
    sentences[i] = ' '.join(words)
```

Responsible for converting sentences to word

lemmatizing the words which are not stopwords.

Que. when to use lemmatization and when to use stemming?

→ If the ultimate goal is to understand the meaning of the text, lemmatization is the better choice.

If the goal is simply to extract features from the text, stemming can be more efficient option.

(*) Bag of Words :-

sent-1 → He is a good boy.
sent-2 → she is a good girl.
sent-3 → Boy & girl are good

stop words ⇒ lowering sentence

sent 1 → good boy
sent-2 → good girl
sent-3 → boy girl good.

words →	good	boy	girl
freqn →	3	2	2

} sort the frequency in desc. order if not sorted.

Binary Bag of words. BOW vectors

	f1 good	f2 boy	f3 girl
sent 1	1	1	0
sent 2	1	0	1
sent 3	1	1	1

* Problems with Bag of words :-

(1) It doesnot tells us about the importance of the words / features.
(2) Not preferrable for larger datasets.
(3) Does not take into account the order of words in a document.
(4) Treats all words as equally important.
(5) computationally expensive.

*) Practical implementation of Bag of words:-

```
import nltk
paragraph = " "

# cleaning the text...
import re
from nltk.corpus import stopwords  <- for removing stop keywords.
from nltk.stem import PorterStemmer
from nltk.stem import WordNetLemmatizer
stemmer = PorterStemmer()  <- for stemming
lemmatizer = WordNetLemmatizer()  <- for lemmatizing

# converting paragraph into sentences...  (converting para to sent.)
sentences = nltk.sent_tokenize(paragraph)
corpus = []  <- for storing cleaned sentences.
for i in range(len(sentences)):
    review = re.sub('[^a-zA-Z]', ' ', sentences[i])  (Replacing all characters other than a-z or A-Z with ' ')
    review = review.lower()
    review = review.split()  <- for converting sent to list of words.
    review = [stemmer.stem(word) for word in review if word not in set(stopwords.words('english'))]
                (Stemming the words other than stopwords.)
    review = ' '.join(review)
    corpus.append(review)
# Creating the Bag of words model...  (converting into counted vector)
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer()
X = cv.fit_transform(corpus).toarray()  -> for converting into matrix form.
```

(*) TF - IDF (Term Frequency - Inverse Document Frequency)

sent 1 → good boy
sent 2 → good girl
sent 3 → boy girl good

→

words	good	boy	girl
freqn	3	2	2

TF = No. of repitition of words in a sentence / No. of words in a sentence

IDF = log (No. of Sentences / No. of sentences containing that word)

TF

Term Frequency			
	sent 1	sent2	sent3
good	1/2	1/2	1/3
boy	1/2	0	1/3
girl	0	1/2	1/3

*

IDF

IDF	
words	IDF
good	log(3/3) = 0
boy	log(3/2) =
girl	log(3/2)

TF-IDF

	f1	f2	f3
	good	boy	girl
sent 1	0	1/2 log(3/2)	0
sent 2	0	0	1/2 log(3/2)
sent 3	0	1/3 log(3/2)	1/3 log(3/2)

*) Practical implementation of TF-IDF :-

```
import nltk
paragraph = "  "
# cleaning---
import re
from nltk.stem import PorterStemmer
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords

stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()

# convert para to sent...
sentences = nltk.sent_tokenize(paragraph)
corpus = [ ]  <- for storing cleaned sentences...

for i in range(len(sentences)):
    review = re.sub('[^a-zA-Z]', ' ', sentences[i])
    review = review.lower()
    review = review.split()  <- conv. sent to words...
    review = [stemmer.stem(word) for word in review
              if word not in
              set(stopwords.words
              ('english'))]
    review = ' '.join(review)
    corpus.append(review)

# creating the TF-IDF model...
from sklearn.feature_extraction.text import TfidfVectorizer
tfid = TfidfVectorizer()
X = tfid.fit_transform(corpus).to_array()
```

we can use any of them as per the requirements

stemming

convert into TF-IDF table/matrix.

(*) WORD EMBEDDING (Feature Representation).

	Boy	Girl	king	Queen	Apple	mango
Gender	-1	1	-0.92	0.93	0.0	0.1
Royal	0.01	0.02	0.95	0.96	-0.02	0.01
Age	0.03	0.04	0.7	0.6	0.92	0.93
⋮	⋮	⋮	⋮	⋮	⋮	⋮
300 dim						

[300 x 1000] → Low Dimension and Dense Matrix.

(*) Recommendor System (*)

*) Key problems:

(1) Gathering "known" ratings for matrix

(2) Extrapolate unknown ratings from the known ones
↳ interested in what you like instead of what you don't like.

(3) Evaluating extrapolation methods.
↳ How to measure performance of recomm. model.

(I) Gathering ratings:-

a) Explicit:
↳ ask users to rate items.
Problem: only small fraction of users leave ratings and reviews.
↓
doesn't scale
So, data not sufficient, or excellent.

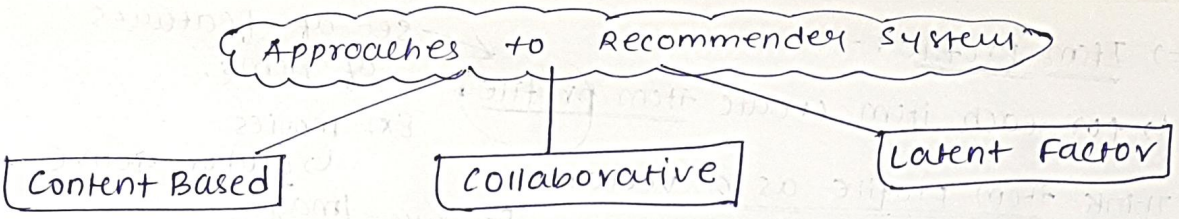
(b) Implicit:-
↳ learn ratings from user actions.
e.g. purchase → high rating
But we cannot capture the low rating using this method.

NOTE: Most of the recomm. system uses combination of explicit and implicit ratings.
When explicit ratings are available they use them and supplement implicit rating with them and use it.

(II) Extrapolating utilities:-

Key problem: matrix U is sparse
as most of users haven't leave ratings and reviews.

Cold start problem:-
↳ New items have no ratings.
↳ New user have no history.



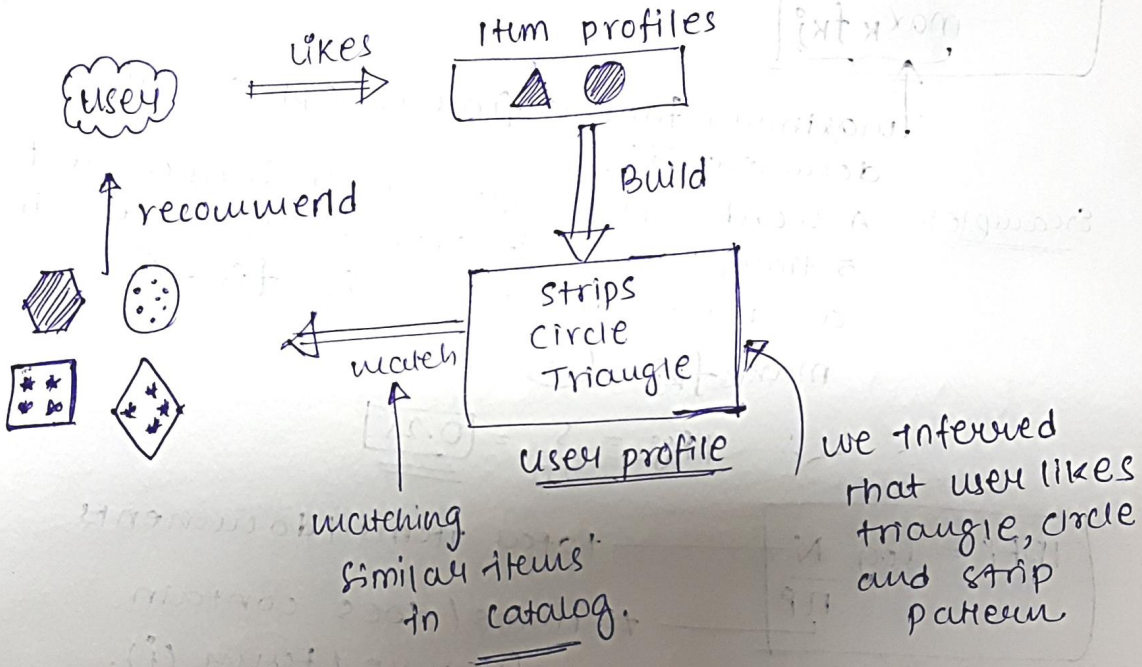
(*) Content Based Recommendation System :-

- **Objective :-** Recommend items → customer (X)
  Similar items to previous items rated highly by customer (X)

Example :- movies
↳ same genre

Blogs
↳ same author, content, etc..

↳ Firstly, we will find set of items user likes using implicit or explicit data.



(=) Item Profile :-

↳ for each item create item profile ← set of features of items.

Ex: movies
↳ author, genre

Think item profile as a vector.
↳ ~~each~~ one entry for each feature. Image
↳ tags, metadata
↳ It might be boolean or real valued.

Here, Profile
↳ set of important words in item document

How to get them?
↳ using **TF-IDF**.

(###) TF-IDF :-

f_{ij} = frequency of feature (i) in item/doc (j)

$$TF_{ij} = \frac{f_{ij}}{\max_k f_{kj}}$$

$\max_k f_{kj}$ ← maximum time a feature appear in document (j).

Example: A word apple appears in a document (j) 5 times. But it appears 20 times in another document. So $f_{ij} = (5)$.

$\max_k f_{kj} = (20)$

$\therefore TF_{ij} = \frac{5}{20} = \boxed{0.25}$

$$IDF_i = \log \frac{N}{n_i}$$

N ← total items / documents.

n_i ← # items / docs contain feature / term (i).

TF-IDF score: $\boxed{w_{ij} = TF_{ij} \times IDF_i}$

NOTE:- Need to Normalize TF to discount for "longer" document.

The more the common term is → larger the IDF → lower the IDF;

NOTE:-
IDF gives lower weights to more common words and higher weights to less common or rarer words.

⇒ create TF-IDF vector, sort it and put some threshold.

⇒ Doc profile :- set of words with highest TF-IDF scores, together with their scores.

(⇒) User Profile :-

↳ user has rated items with profile $i_1, i_2, \ldots, i_n$.

↳ simple average of rated item profiles.

↳ Problems :- Doesn't take into account the likeness of item.
It takes all items to be equally likely.

Solution :- Take weighted average.

Variant :- Normalize weights using average rating of user.

Example :- Boolean utility matrix :

items → movies
feature → "Actor"

item profile → vector of 0 or 1 for each actor.

X has watched 5 movies.
2 consisting of Actor A → weight = 2/5 = 0.4
3 " of actor B → weight = 3/5 = 0.6.

Example :- star ratings.

Actor A's movies rated 3 and 5
Actor B's movies rated 1, 2 and 4

(*) Normalization helps us to capture the idea that some are the +ve ratings and some of them are -ve ratings.

(*) users are diff. their perspective might be diff. some consider 4 as ~~average~~ avg rating and some consider 4 as +ve rating (highly).

⇒ Normalizing ratings by subtracting user's mean:-

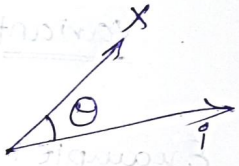
Actor "A" :- 0, +2 → weight = (0+2)/2 = (1)

Actor "B" :- -2, -1, +1 → weight = (-2/3)

(=) Making Predictions :-

Taking pair of user profile (X) and item profile(i) and finding cosine similarity bet^n them.

$$U(X,i) = \cos(\theta) = (X \cdot i)/(|X| \cdot |i|)$$



(*) Smaller the (θ) higher the similarity.

NOTE:- cosine distance is the actually angle (θ). and cosine similarity is the angle (180-θ). For our convience, we use cos(θ) as our similarity measure and call it cosine similarity...

→ No need ~~of data~~ for data on other users.

→ Able to recommend to users with unique tastes
↳ in collaborative filtering it is difficult to find similar users with unique taste.

→ Able to recommend new and unpopular items
↳ No first-rater problem.

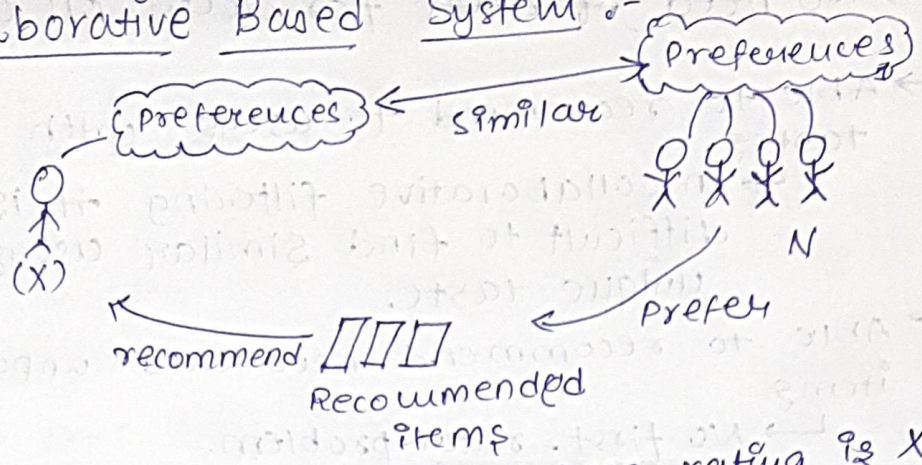
→ Explanations for recommended items
↳ we can explain user that based on what feature the given item is recommended...

→ Finding the appropriate features is hard.

→ Overspecialization
- Never recommends items outside user's content profile.
- people might have multiple interests
- unable to exploit quality judgements of other users.

→ cold-start problem for new users.
↳ How to build user-profile?

(*) Collaborative Based System :-



⇒ Find set N of other users whose rating is X's rating.
⇒ Estimate X's ratings based on ratings of users in N.

Example:-

	HP1	HP2	HP3	TW	SW1	SW2	SW3
A	4			5	1		
B	5	5	4				
C				2	4	5	
D		3					3

→ r_A, r_B are the respective rating vectors for users A and B.
→ we need similarity metric $sim(A,B)$.
→ Here we can see $sim(A,B) > sim(A,C)$.

Option-1 :- Jaccard similarity.

$$sim(A,B) = |r_A \cap r_B| / |r_A \cup r_B|$$

$sim(A,B) = 4/5$
$sim(A,C) = 2/4$

$sim(A,B) < sim(A,C)$ it counters the notion that we have captured intuitively.

Problem :- ignores rating values...

- Option-2:- cosine similarity :-

$Sim(A,B) = \cos(r_A, r_B)$ → 0.38

$sim(A,C) = 0.32$

} $sim(A,B) > sim(A,C)$ but not by too much.

⇒ we had filled empty ratings with zero.

Problem:- treats missing ratings as negative.

- Option 3:- centred cosine: (also known as Pearson correlation)

→ Normalize the rating by ~~centred~~ subtracting row mean.

avg. rating of A = 10/3 = 3.33

avg. rating of B = 14/3

" of C = 11/3

" of D = 6/2 = 3

Modified:

	HP1	HP2	HP3	TW	SW1	SW2	SW3
A	2/3			5/3	-7/3		
B	4/3	4/3	-2/3	-5/3	1/3	4/3	
C							0
D		0					

} sum of ratings row wise = 0

zero becomes avg. rating of each user.

Now, $sim(A,B) = \cos(r_A, r_B) = 0.09$

$sim(A,C) = -0.56$

$\therefore sim(A,B) > sim(A,C)$

☺ captures intuition better
- ↳ missing ratings treated as "average".
- ↳ Handles "tough raters" and "easy raters".

(=) Rating Predictions :-

→ Let r_x be the user x's ratings.

→ Let N be the set of K most similar to X users. who have also rated item i

→ Prediction for user x and item i

→ set N consist only those users who have rated item (i).

Option 1 :- Average Rating from the neighbours.

$$r_{xi} = 1/K \sum_{y \in N} r_{yi}$$

Here, all users (neighbours) given equal importance.

Option 2 :- weighted Average

$$r_{xi} = \frac{\sum_{y \in N} S_{xy}\, r_{yi}}{\sum_{y \in N} S_{xy}}$$

S_{xy} ← sim(x,y)

(*) Item-Item Collaborative Filtering :-

1) For item i, find other similar items.
2) Estimate rating for item i based on ratings for similar items.

(*) Can use the same similarity metrics and prediction functions as in user-user model.

$$r_{xi} = \frac{\sum_{j \in N(i;x)} s_{ij} \cdot r_{xj}}{\sum_{j \in N(i;x)} s_{ij}}$$

- r_{xj} → rating of user x on item j
- s_{ij} → similarity of items i and j
- $N(i;x)$ → set of items rated by x similar to i.

Example :- Item-Item CF ($|N|=2$) → No. of neighbours = (2).

movies \ users	1	2	3	4	5	6	7	8	9	10	11	12	Sim(1,m)
1	1		3		?	5			5		4		1.00
2			5	4			4			2	1	3	-0.18
3	2	4		1	2		3		4	3	5		0.41
4		2	4	3	5	2		4			2		-0.10
5			4		4								
6	1		3		3			2			2	5	-0.31
											4		0.59

Here we use Pearson ~~cosine~~ correlation as similarity :

(1) subtract the mean rating m_i, for each movie i

$m_1 = (1+3+5+5+4)/5 = 3.6$

row 1 = $[-2.6, 0, -0.6, 0, 0, 1.4, 0, 0, 1.4, 0, 0.4, 0]$

(2) compute the cosine similarities between rows...

$s_{13} = 0.41$ $s_{16} = 0.59$

Predict by taking weighted average :-

$$r_{15} = \frac{(0.41 * 2 + 0.59 * 3)}{(0.41 + 0.59)} \cong \boxed{2.6}$$

(*) Item-Item vs User-User :-

→ in practice Item-Item outperforms user-user in many use cases. like movie, books, etc..

Why?

↳ Item-Item are "simpler" than users.

↳ Item belongs to small set of "genres", users have varied ~~tests~~ tastes.

↳ Item similarity more meaningful than user similarity.

(*) Implementing Collaborative Filtering :-

(=) Complexity :-

↳ most expensive step is finding K-most similar user or items → $\boxed{O(|U|)}$ $|U|$ ← size of utility matrix.

→ To expensive to do at runtime

↳ could pre-compute

↳ Naive pre-computation takes time

$O(n \cdot |U|)$ $n = \#$ ~~us~~ items / users

Ways :-

→ Near-neighbour search in high dimensions.

→ Clustering ← we can restrict the incluster searching to save time.

→ Dimensionality reduction.

(=) Pros/Cons of Collaborative Filtering :-

- works for any kind of item. } It is tough to find right set of features
  ↳ no feature selection needed. }

- Cold start Problem. Need enough no. of users to find a match
- Sparsity. - user/ratings matrix is sparse.
  - Hard to find users that have rated same items.
- First rater Problem :- - cannot recommend an unrated item.
  - New items, Esoteric items.
- Popularity Bias :-
  - Tend to recommend popular items.

(=) Hybrid Methods :-

⇒ Add content based methods to collaborative Filtering
- Item profiles for new item problems.
- Demographics to deal with new user problem.

⇒ Implement two or more different recommenders and combine predictions.
- using a linear model.
- Ex:- global baseline + collaborative filtering.

(=) Global Baseline Estimates :-

→ Estimate Hiren's rating for movie KGF2
Problem : Hiren has not rated any movie "Similar" to KGF2.

Global Baseline Approach :
- Mean movie rating :- 3.7 stars.
- KGF2 is 0.5 stars above avg.
- Hiren's rating is 0.2 below the avg.
- Baseline estimate :- 3.7+0.5-0.2 = 4 stars.

(*) Haven't used any information about the movies Hiren rated. Similar to KGF2.

(≡) Combining Global Baseline with CF :-

⇒ Global Baseline estimates
- Hiren will give KGF2, 4 stars.

⇒ Local Neighbourhood (CF/NN) :-
- Hiren didn't like related movie KGF.
- Rated it 1 star below the avg. rating

⇒ Final estimate.
- Hiren will rate KGF2 4-1 = **3 stars**

- **Common Practice :-**
  - Define similarities s_{ij} of items i and j
  - select the K nearest neighbours $N(i;x)$
    items most similar to i and rated by user (x).
  - Estimate rating r_{xi} as the weighted avg :

$$r_{xi} = b_{xi} + \frac{\sum_{j \in N(i;x)} s_{ij}(r_{xj} - b_{xj})}{\sum_{j \in N(i;x)} s_{ij}}$$

Baseline estimator for r_{xi}

$$b_{xi} = \mu + b_x + b_i$$

overall mean movie rating

b_x = rating deviation of user x
= (avg. rating of user x) − μ

b_i = rating deviation of movie (i).

(*) Evaluating Recommender Systems :-

- Take the piece of data from the user-item rating matrix. and consider as test set.
- compare the predictions against withheld known ratings
- Root mean squared Error

$$\sqrt{\frac{\sum_{(x,i)\in T} (r_{xi} - r_{xi}^*)^2}{N}}$$

$N = |T|$
$r_{xi}^* \leftarrow$ actual.
$r_{xi} \leftarrow$ predicted.

(*) Problems with Error Measures :-

⇒ Narrow focus on accuracy sometimes misses
- Prediction Diversity.
- Prediction content.
- order of predictions.

⇒ we care only to predict High ratings (in practice)
↳ RMSE might penalize a method that does well for high ratings and badly for others.

Alternative: precision at top K.
- % of predictions in the user's top K withheld ratings.