Generative AI for Developers
Comprehensive Course

- freecodecamp.org
- 21 hour long video

* Gen AI

End to End pipeline

* Set of steps followed to build an end to end GenAI software

* Data acquisition
Data Preparation
Feature Engineering
Modeling
Evaluation
Deployment
Monitoring and model updating

(1) Data Acquisition

* Available data (csv, txt, PDF, Docs, xlsx)

* Other data (DB, Internet, API, Scrapping)

* No Data (create your own data)
(LLM (Generate from them))

Note:- If have data augment the data

Ex:- ① Replace with synonyms

→ I am a data scientist
augmented → I am a AI engineer.

② Bigram flip
→ I am Rohit
→ Rohit is my name

③ Back translate
* translate the lines in many languages and get it back to original language in this way something will be changed.

④ Add Additional Data / Noise

I am a data scientist, I love this job
→ Additional data

(2) Data Preprocessing

(1) Cleanup: HTML, emoji, spelling correction ← as now we also use these

(2) Basic Preprocessing

(3) Advance Preprocessing

Basic

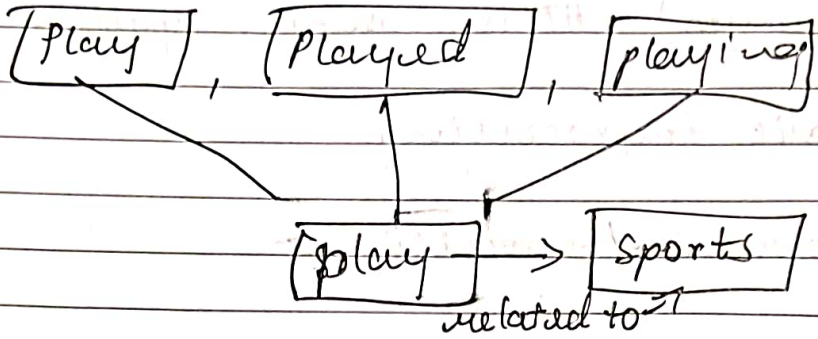
Tokenization — Sentence level
— word level

~~Stemming~~ ["my", "name", "is", "Rohit"] word level

~~Lemmatizing~~ ["I am Rohit", "I am engineer"] sentence level

optional preprocessing
(1) Stop word removal
(2) Steaming — less used
(3) Lemmatization — more used
(4) Punctution removal (?, -, !, $)
(5) lower case
(6) language Detection

* what is streaming?



It reduces dimensions

* What is lower casing?

Rohit is human
rohit is Engineering

* as human we know Rohit
rohit is same
* but model think both
are same so we
convert to lower case
to avoid confusion

Advance Pre processing

1. Parts of speech
2. Parsing
3. Corefference Resolution

Ex:- Input

Rohit wrote, directed, and composed the music of most of his films.

Tokenization with lemmatization

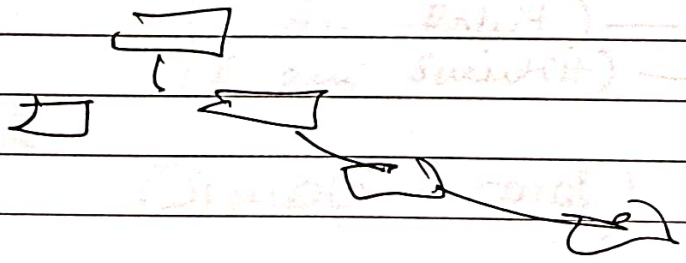
Rohit write is direct , compose the music for
Rohit wrote , directed, composed the music for
most of he film .
most of his films .

POS tagging

NNP VBD , VBD CC VBN DT NN
Rohit wrote , directed and composed the music
IN JJS IN PRP$ NNS .
for most of his films .

* here we are mentioning what parts of speech noun, pronoun, verb, etc

Parse tree



Coreference Resolution

mention — — — . coref — — — — — — . . mention
Rohit wrote, directed . . — — — — — his films

* means Rohit, his is same

(3) Feature Engineering

text vectorization
- → TFIDF
- → Bag of word
- word 2 vec
- one hot
- Transformers model → used for advance like llms

(4) Modeling

Can choose different models
- open source LLMs (need good gpu, memory, etc)
- paid model (used as cloud gpus, etc)
  - open AI

(5) Evaluation

① Intrinsive — metrin

② Extrinsive — Deployment
like chatgpt asks feedback
in between

(6) Deployment

monitoring and retraining

Terms you must Remember

① Corpus
*(Entire text)
(I am Rohit) corpus

② Vocabulary
*(unique word)
→ I unique
→ am unique
→ Rohit unique

③ Documents
* (one row/line)
(I am Rohit) → document
(I am Scientist) → documents

④ word
(Simple word)

Data preprocessing

* Lets use IMBD dataset from kaggle with 50k rows

it is csv format
so can use pd

has 3 columns
index, review, sentiment

⇒ data_path = "/————————/"

df = pd.read_csv(data_path)

* lets do lower casing

df['review'] = df['review'].str.lower()

↘ will convert everything to lower case

Html tags remove

→ import re
def remove_html_tags(text):
    pattern = re.compile('<.*?>')
    return pattern.sub(r'', text)

this is a class which will remove < > tags

→ remove_html_tags("<p> Hi <p>")

→ 'Hi'

→ df['r___'] = df['r___'].apply(remove_html_tags)

remove url

→ def remove_url(text):
    pattern = re.compile(r'https?://\S+|www\.\S+')
    return pattern.sub(r'', text)

will remove urls from text

Punctuation handling

→ Import string
string.punctuation

→ include = string.punctuation
include
→ '!"#$%&...

→ def remove_punc(text):
    for char in include:
        text = text.replace(char, '')
    return text

↳ we are replacing $, ?, ! with space

but this will take time for lot of data

so

→ def remove_punc(text):
    return text.translate(str.maketrans('', '', include))

→ df['r——'] = ~~df~~ remove_punc(df['r——'])

Handle chat conversation

```
chat_words = {
        'AFAIK': 'A Far As I know',
        'ASAP': 'As soon as Possible',
        .
        .
        .
        }
```

* These are shortcuts we use in chats and social media so we need to handle there

* What we can do is map them to real meaning

→
```
def Chat_conv(text):
    new_text = []
    for w in text.split():
        if w.upper() in chat_words:
            new_text.append(chat_words[w.upper()])
        else:
            new_text.append(w)
```

return " ".join(new_text)

→ Chat_conv ('Do it ASAP')
→ 'Do it As Soon as Possible

* Incorrect Text handling

→ from textblob import Textblob

→ incorrect_text = 'certain conditionas'

textblob = Textblob(incorrect_text)

textblb.correct().string

→ 'certain Conditions'

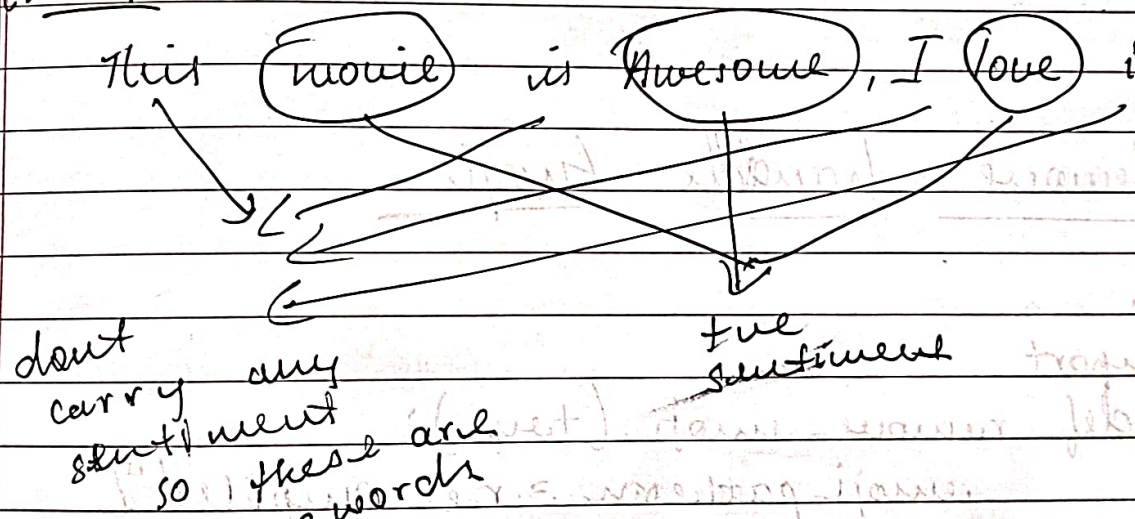
* Stopwords

→ from nltk.corpus import stopwords
import nltk
nltk.download ('stopwords')

→ stopwords.words('english') → as it has many langs and we need eng

→ [i
me
my
myself
we
.
.
.
]

Example :-



→ def remove_stopwords(text):
    new_text = []

    for word in text.split():
        for word in stop

* generate through gpt

→ df['r___'] = df['re___'].apply(remove_stopwords)

* Remove - handle emoji

→ import
def remove_emoji(text):
    emoji_pattern = re.compile("["
        u"\U00010F600-\U0001F64F"

"]+", flags = re.UNICODE)
return emoji_pattern.sub(r'', text)

* UNICODE

is a code which every emoji has by. that we remove it.

* But now we see that our models can handle emojis so we can keep it through process

→ import emoji

print (emoji.demojize ('Python is 🔥'))

→ python is :fire:

↳ Now we see our emoji is textualized and can be processed.

* Tokenization

→ sent1 = 'I am in Hyd'

sent1.split()

→ ["I", "am", "in", "Hyd"]

* word level

→ Sent2 = 'I am in Hyd. Going to daman'

~~split~~ Sent2.split('.')

→ ['I am ——— Hyd', " Going ——— "]

* Regular Expression

→ import re
Sent3 = 'I am in delhi'
tokens = re.findall("[\w']+", sent3)
tokens
→ ['I', 'am', 'in', 'dlhi']

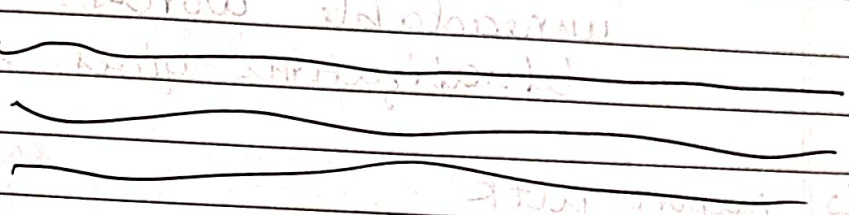
for ~~word~~ word level

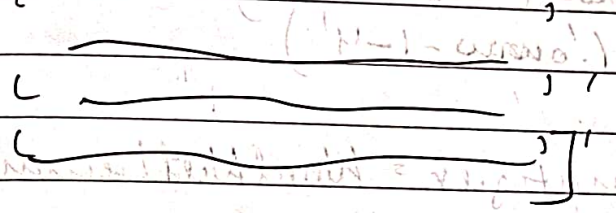
* NLTK way to tokenize

→ from nltk.tokenize import word_tokenize, sent_tokenize

import nltk
nltk.download('punkt')

→ sent1 = 'I am Rohit'
word_tokenize(sent1)
→ ['I', 'am', 'Rohit']

→ text = '

'

sent_tokenize(text)
→ ['

']

* Stemmer

Play Played Playing
↓
Play (to root form)

→ from nltk.stem.porter import PorterStemmer

sample = "walks walking walked"
PorterStemmer(sample)
→ 'walk walk walk'

Lemmatization

* Same like Stemming but stemming gives unreadable words
  lemmatization gives readable

→ import nltk
from nltk.stem import WordNetLemmatizer
import nltk
nltk.download('wordnet')
" . " ('omw-1-4')

wordnet_lemmatizer = WordNetLemmatizer()

→ sentence = "
punctuations = "?:!.,;"

sentence_words = nltk.word_tokenize(sentence)
for word in sentence_words:
    if word in punctuations:
        sentence_words.remove(word)

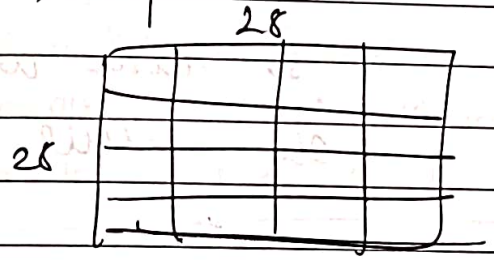
generated GPT

* Lemmatization is slow and stemming is fast

Data Representation

1. what is feature extraction from text/image
2. why we need it?
3. why it is so difficult?
4. what is the core idea
5. Some techniques

* As for Images



they are pixels and values

28 x 28 pixels means 784 pixels so 784 input neurons

* for Audio

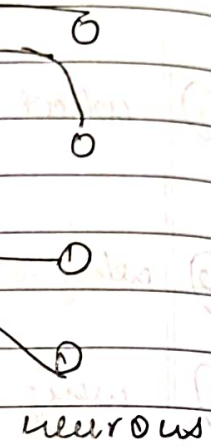
we give frequency and dB as input

and fit it

As for text data:

① My name is Buppy

0 1 2 3



② How are you

4 5 6

it expects 4 dimension input but we give 3 dim input

so there will be dim issue

+ For that we can do some techniques

1. One hot encoding
2. Bag of word (BOW)

One hot encoding

Lets assume we are give 4 data sentences

D_1	People watch dswithbuppy
D_2	dswithbuppy watch dswithbuppy
D_3	people write comments
D_4	dswithbuppy write comments

we have total 5 unique words

people	watch	dswithbuppy	write	comments
1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
0	0	0	1	0
0	0	0	0	1

So for D_1 the vector is

$$D_1 = [[[1\ 0\ 0\ 0\ 0],\ [0\ 1\ 0\ 0\ 0],\ [0\ 0\ 1\ 0\ 0]]$$

this is the sentence 1

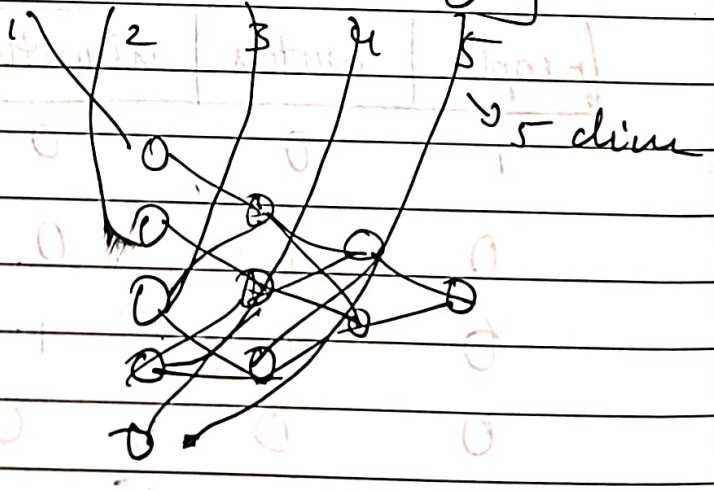
$$D_2 = [[[0\ 0\ 1\ 0\ 0],\ [0\ 1\ 0\ 0\ 0],\ [0\ 0\ 1\ 0\ 0]]$$

for sentence 2

Basically it give vector for every word position

we can see 5 dim vector

$$[0\ 0\ 1\ 0\ 0]$$



In case the vocab size means unique words is 2000

the input size becomes 2000

so it is not recommended

the computation cost will ↑

and one more problem

new sentence [My name is Bappy]

these words are not there in the corpus

out of vocabulary issue (OOV)

Another issue, we have lots of zeros

sparsity issue

0 is unnecessary number just increasing computation

Drawbacks

1. Sparsity
2. No fixed size
3. OOV
4. Not capturing semantic meaning

Bag of Word

* Let consider the same previous corpus.

* What this here is unlike the word position, here word count is done

pro—	watch	dsw—	write	comment
D_1 = [1	1	1	0	0]
D_2 = [0	1	2	0	0]
D_3 = [1	0	0	1	1]
	[0	0	1	1	1]

* We see less zeros than One hot

* Research says we can use it for sentiment analysis

Drawbacks

1. It is not capturing semantic info
2. Still has few zeros

* We have more techniques
  * TFIDF
  * Word2Vector
  * Transformers

Practical for BOW

→ import numpy as np
import pandas as pd

→ df = pd.DataFrame({"text": ["p____", "dsv____", "____", "____"], "output": [1, 1, 0, 0]})

df

→

	text	output
0	~~~~	1
1	~~~~	1
2	~~~~	0
3	~~~~	0

this is sentiment +ve or -ve

→ from sklearn.feature_extraction.text import CountVectorizer

cv = CountVectorizer() → BOW technique

→ bow = cv.fit_transform(df['text'])

print(cv.vocabulary_)

→ {'proper': 2, 'world': 3, 'is': 1, 'a': 0}

→ it will assign indexing for the vocab

→ bow.toarray()

→ array([[0, 1, 1, 1, 0], # d1
[0, 2, 0, 1, 0], # d2
[1, 0, 1, 0, 1], # d3
[0, 1, 0, 0, 1]]) # d4

→ X = bow.toarray()
y = df['output']

N-grams

df = pd.D—— ({ 'text': ——

——

——

})

* Means there will be pairs of 2 word
  Bi-gram
  if pair of 3 words
  tri-gram

* people watch
  watch ds——

* This helps capturing the sentiment better

* This is good movie (+ve)
* This is not good movie (-ve)

↳ we see not good is found means caputures -ve sentiment

→ cv = CountVectorizer(ngram_range(2,2))

↓

mean will consider 2 words bi-gram

print(cv.vocabulary_)

→ {'people watch': 2, 'watch dsmith': 4, }

TF-IDF Practical

TF - IDF

Terms x within document y

formula

$$w_{x,y} = tf_{x,y} \times \log\left(\frac{N}{df_x}\right)$$

tf = frequency of x in y

df = no of documents containing x

N = total no. of components

this the formula used

→ from sklearn.feature_extraction.text import TfidfVectorizer

tfid = TfidfVectorizer()

→ arr = tfid.fit_transform(df['text']).toarray()
arr

→ array([[0. , 0.4968 , 0.61366, 0.61366, 0. ,
[0. , 0.8504..., 0. , 0.5254..., 0.]
[ ]
[ ]])

* this is better than BOW as has lesser sparsity
* as even Tfid is not able to contain semantic info

Deep learning method

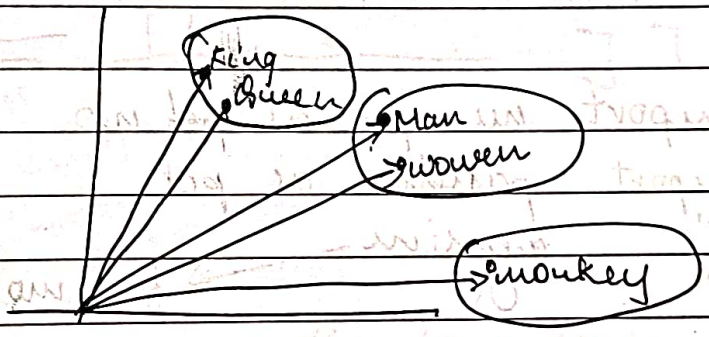
* Word2Vec technique

Lets assume some creatures and some of their features

features	king	Queen	Man	Women	Monkey
gender	1	0	1	0	1
wealth	1	1	0.3	0.2	0
power	1	0.7	0.3	0.2	0
weight	0.8	0.5	0.7	0.5	0.3
speak	1	1	1	1	0

Lets assume some initial values

lets represent this in a 2D space



we see 3 different clusters

as king and Queen are related
Man and women related
a monkey in a separate Cluster

if we add a sentence princess is hungry
where will it go ofcourse the the king Cluster

In this way we use Word2Vec capture sentiment

The feature are generated by the model neural network
These we discussed were just for understanding
This way the model will learn to cluster and find sentiments

Practical

We are gonna use game of thrones dialogue dataset from kaggle

→ import numpy as ~~pd~~ np
import pandas as pd
" gensim → for model
" os

→ !pip install --upgrade gensim --user

→ from nltk import sent_tokenize
" gensim.utils import simple_preprocess → here we have some base pre processing
import nltk
nltk.download('punkt')

→ story = []
for filename in os.listdir('data'):

(generate GPT

* this will extract all the text into story will use sent_tokenize and tokenize the sentences

story

→ [[game,
of ,
,] [. . .]]

→ len(story)
→ 8602

→ story[0] # means the first dialogue

['game',
'of'
'thr',
]

initialize model

→ model = gensim.models.Word2Vec(
window=10
min_count=2
)

→ model.buil_vocab(story)

↳ converting into vector representation

→ model.train(story, total_examples = model.corpus_count, epochs = model.epochs)

↳ training

→ model.wv.most_similar('daenerys')

↳ we are seeing which words are more related to this word

([('still', 0.99...
'fire', 0.99...
...])

→ model.wv.similarity('arya', 'sansa')
→ 0.99974376
↳ relationship

→ vec = model.wv.get_normed_vectors()
↳ this will give all the vectors

Lets do its visualizations

Lets do PCA 1st to reduce dimensions
→ from sklearn import PCA
PCA = PCA(n_components=3)

→ x = pca.fit_transform(model.wv.get_normed_vector())

→ import plotly.express as px
fig = px.scatter_3d(x[200:300], x=0, y=1, z=2, colour=y[200:300])

fig.show()

→ # we use a good 3d plotting to visualize the scatters and clusters

* OPENAI API

* How IT WORKS

* ChatCompition Model
  and Compltion
  Model

* Hugging face
  Transformers

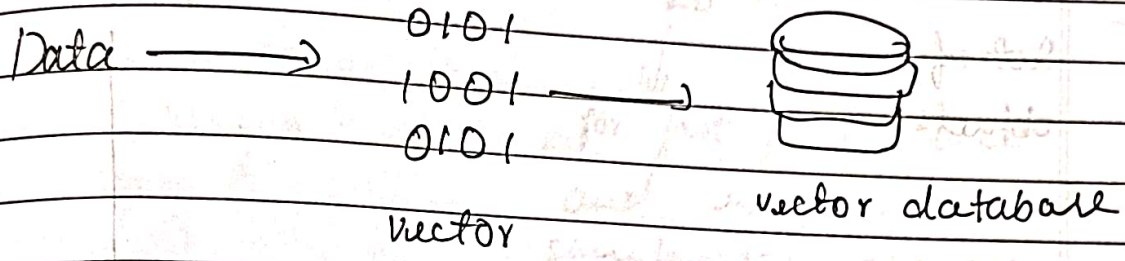
* Some Projects
  Whisper
  DALLE

* Prompt Enginering Mastery

* watch in video
  easy

Vector Database

whatever data we are using to send to a model should be converted to VD



Use langchain for this.

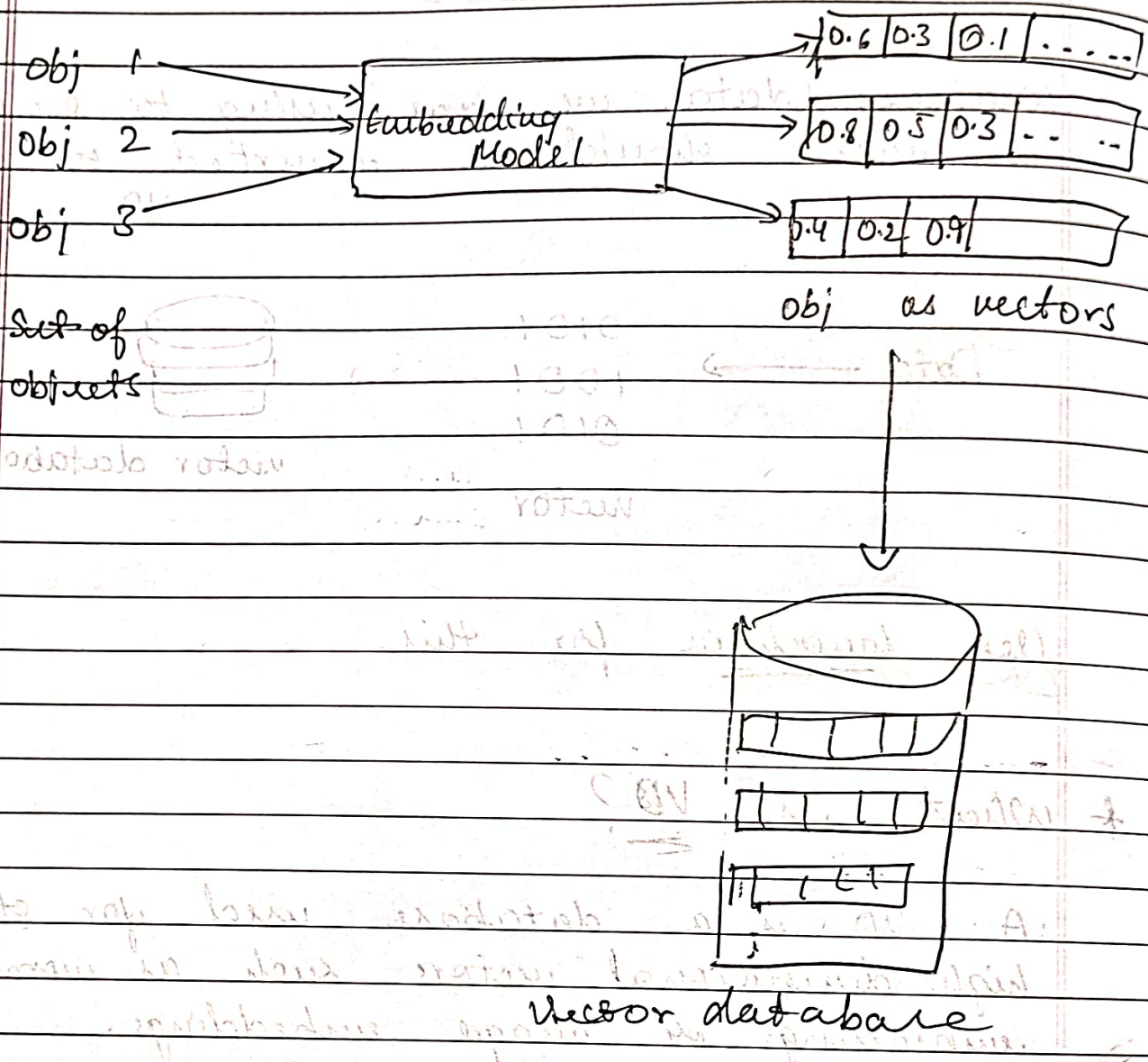
* What is VD?

A VD is a database used for storing high dimensional vectors such as word embeddings or image embeddings.

Why?

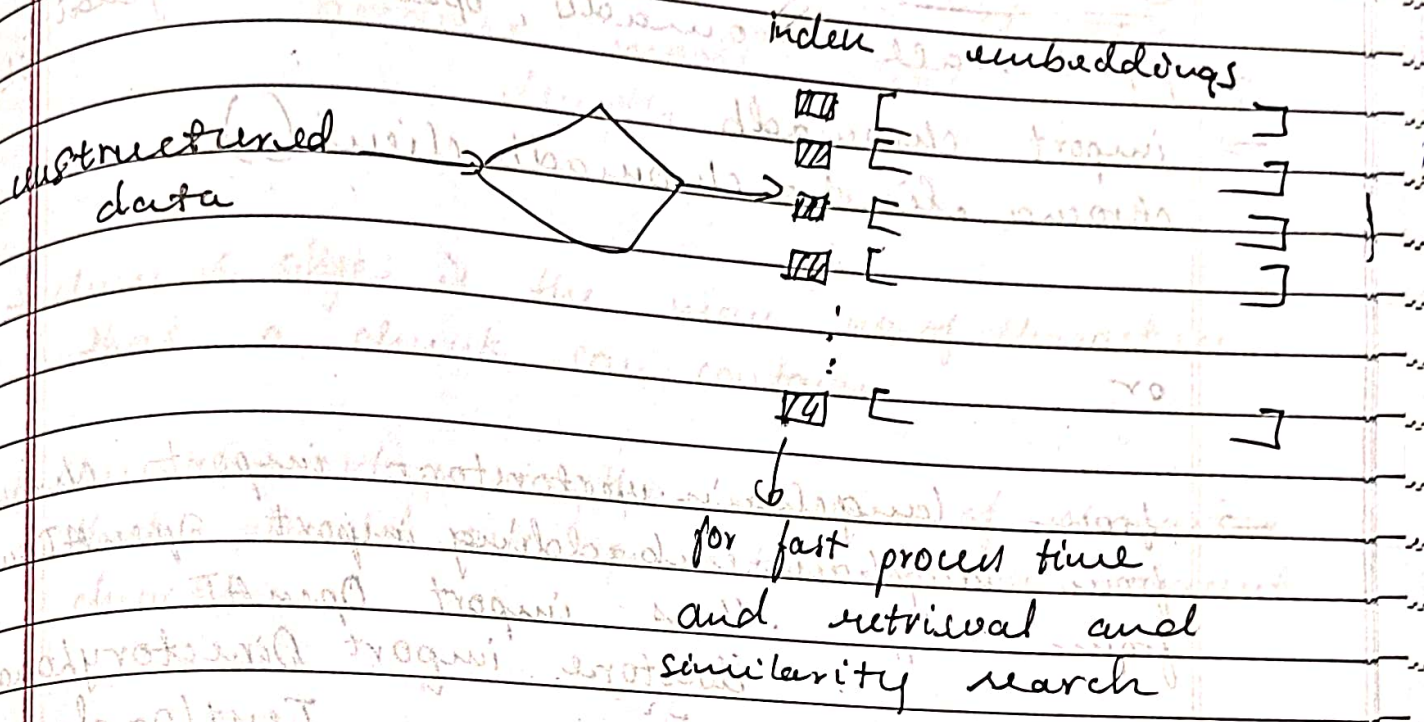
Over 80-85% data out there is unstructured data, we can't easily store them into a Relational/Traditional database!!

Vector Embeddings



* We have lots of embedding Models
* OPENAI embedding
* Huggingface emb
* Llama 2 emb
* Google palm emb

Vector Indexes



* Some vector DB

ChromaDB
Weaviate
Pinecone
neo4j

Chroma DB

→ pip install chromadb, openai langchain tiktoken

→ import chromadb
chroma_client = chromadb.Client()

or

→ from langchain.vectorstore import Chromadb
from langchain.embedding import OpenAIEmbeddings
from ".llms import OpenAI
" ".docstore import DirectoryLoader
TextLoader

#load data

loader
→ Directoryloader("<-----<--" glob = "./*.txt"
, loader_cls = TextLoader)

→ document = loader.load()

loaded data Succesfully

* All the data is in big para we need to break into chunck because model has a input length.

* chunck size is the max no. of characters that a chunk can contain.

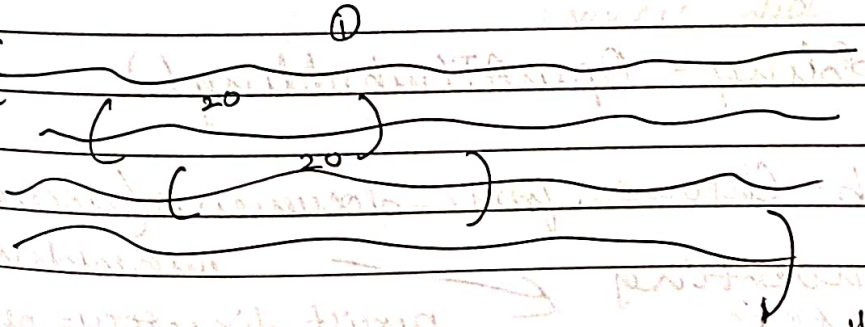
* Chunk overlap is the number of character that should overlap between two adjacent chunks

Ex:- chunk-size = 500

* will take 500 char and stop and continue separating the whole corpus into 500 char size chunks.

* as per open ai embeddings input size is less than 4095 tokens

Ex: chunk overlap = 20



means will take start point 20 chars before this is useful for knowing prev content

→ from langchain.text_splitter import
RecursiveCharacter
TextSplitter

↓

To convert into chunks

```
text_splitter =
→ RecursiveCharacterTextSplitter(chunk_size=1000,
                                 chunk_overlap=200)

text =
text_splitter.split_documents(document)

len(text)

→ 233
  # chunks
```

Creating DB

```
→ from langchain import embeddings
persist_directory = 'db'

embedding = OpenAIEmbedding()

vectordb = Chroma.from_documents(documents=texts,
                                 embedding=emb
                                 persist_directory=persist_directo
```

we converting chunks to vectors

persiste the db to disk

→ vectordb.persist()
vectordb = None

↘ a folder will be saved in binary representation

Now we want to load from disk

→ vectordb = Chroma(persist_directory = persist_directory, embedding_function = embedding)

Now can apply Similarity Search by Retriever

→ retriever = vectordb.as_retriever()

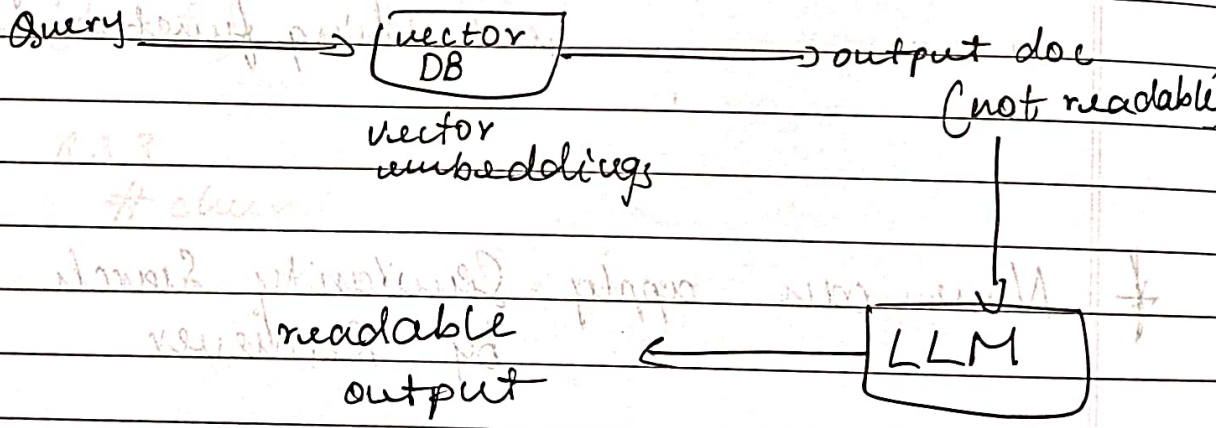
docs = retriever.get_relevant_documents("How much money did Microsoft raise?")

→ docs

→ will return 4 similar documents from the db

* We see that it is giving alot of irrelevant or unnecessary

So we take help from LLM



Make a chain

→ from langchain.chains import RetrivalQA

create the chain to answer questions

qa_chain = RetrivalQA.from_chain_type(llm=OpenAI(),
chain_type="stuff",
retriever=retriever,
return_source_documents=true)

will also give doc source

→ query = "How much did ——— ?"
llm_response = qa_chain(query)
llm_response

→ {'query': ' ——— ',
'result': ' Around $10 billion.'}

Deleting DB

make "zip" file of the db

→ ! zip -r db.zip /db

→ vectordb.delete_collection()
vectordb.persist()

!rm -rf db/

Pinecone VDB

* Install the packages
  langchain
  pinecone-client
  pypdf → to load pdf data
  openAI
  tiktoken

* Libraries
  from langchain.document_loaders import PyPDFDirectoryLoader
  " " .text_splitter " RecursiveCharacterTextSplitter
  " " .embeddings " OpenAIEmbeddings
  " " .llms import OPENAI
  " " .vectorstores imp Pinecone
  " " .chains " RetrievalQA
  " " .prompts " PromptTemplate
  " OS

Load Pdf

* Download the file

* Extract text

→ loader = PyPDFDi——("pdfs") ← folder containing 2 Pdfs

data = loader.load()

→ loads the pdfs texts in it

Convert to chunks

→ text_splitter = Recu——(chunk_size=500, chunk_overlap=20)

text_chunks = text_splitter.split_documents(data)

→ chunks done

Download Embedding

→ os.environ['OPENAI_API_KEY']="sk——"

embeddings = OpenAIEm——()

result = embeddings.embed_query("How are you")

→ 1536

*means this text has generated a vector with 1536 features

* Initializing Pinecone

→ Pinecone_api_key = os.environ.get('PINECONE_API_KEY', '————————as')

Pinecone_api_env = os.'' .''('' '' _ '' '' _ENV', '————————')

this you get by creating index

note: we have mention the dimension of input vector and ours was 1536

→ pinecone.init(
    api_key = Pinecone_api_key
    environment = Pinecone_api_env
)

index_name = 'test'

have to give name same as name of env in pinecone

* Create embeddings for each chunks

→ docsearch = Pinecone.from_texts(
[t.page_content for t in chunks],
embedding, index_name =
index_name)

converted to vectors
and can also see in
pinecone cloud interface

* If already have an index, you can
load it like this

→ docsearch = Pinecone.from_existing_index(
index_name,
embeddings)

docsearch

→ <langchain.vectorstores.pinecone.Pinecone
at 0x7f...>

* Similarity Search

query = "YOLOv7 outperforms which models"
docs = docsearch.similarity_search(query, k=3)
will give 3 out

* Creating LLM model wrapper

→ qa = RetrievalQA.from_chain_type(llm=llm,
chain_type="stuff",
retriever=docsearch.as_retriever())

qa.run(query)

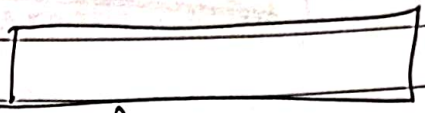
→ YOLOV7

→ we get llm output which is readable

* Lets make it input based query

→ while True:
    user_input = input(f"Input Prompt:")
    if user_input == 'exit':
        print('Existing')
        sys.exit()
    if user_input == '':
        continue
    result = qa({'query': user_input})

print(f"Answer: {result['result']}")

→ Input prompt: 

LANGCHAIN
BASIC TO ADVANCE

* We know How to use the OPENAI with langcha

* What about Hugging face models

→ from langchain import HuggingfaceHub

→ llm = HuggingFaceHub(repo_id = "~~~~~" ,
(repo id of the model)
model_kwargs = {
"temperature": 0,
"max_length": 64 })

llm("Translate to English: How old are you?")

→ ~~~~~

Prompt template and Chain

⇒ llm = OPENAI (temperature = 0.6)

prompt_template_name = PromptTemplate (
    input_variables = ['cuisine'],
    template = "I want
        {cuisine} food.
        Suggest fancy name."
)

this is template of input and input can be a single word like indian

name_chain = LLMChain(llm = llm, prompt =
    prompt_template_name)

prompt_temp_items = PromptTemplate (
    input_variable = ['restaurant_name',
    template = """ Suggest
        {restaurant_name} """)

food_item_chain = LLMChain (llm = llm,
    prompt = prompt_temp_items)

* Now we want the both chains to run after another

→ from langchain.chains import SimpleSequentialChain

→ Chain = SimpleSe——(chains=[name_chain, food_items_chain])

content = chain.run("indian")
↘ input

print(content)

→ we get output

↘ limitation is
we only see the
last prompt output

* to overcome this we use sequential Chain not Simple

→ chain = SequentialChain (chains = [~~~], input_variable = ['cuisine'], output_variable = ['res~~~ ~~~, 'men~~~ '])

→ print (chain ({"cuisine": "indian"}))

→ output

Agents and tools in langchain

* Agent is a powerful concept in langchain

* For Eg:

"I have to travel to Dubai to Canada"

* I type this in ChatGPT

→ "Give me 2 flights options on, September 1, 2024

* GPT will not be able to ans because has knowledge till september 2021

* ChatGPT plus has expedia plugin, if we enable this plugin it will go to Expedia Plugin and will try to pull info about flights and it will show the info

SerpApi is a real-time API to access Google search results.

Ques

→ from langchain.agents import AgentType, Initialize_agent, load_tools

from langchain.llms import OpenAI

* Pip install wikipedia

→ llm = OpenAI(temp = 0)

→ tools = load_tools(["wikipedia", "llm-math"], llm=llm)

tools we are going give access to the agent

→ agent = Initialize_agent(
tools,
llm,
agent = AgentType.ZERO_SHOT_REACT_DISCRIPTIO
verbose: True)

→ # Now lets test the agent

agent.run("What is the GDP of US in 2024?")

→ "—————————— $28.269 trillion"

Memory

* We see that GPT has some memory to Remember

* ConversationBufferMemory

* we can attach memory to remember all prev convo.

→ from langchain.memory import ConversationBufferMemory

memory = Con_____()

chain = LLMChain(llm=llm, prompt=pro_____
, memory=memory)

name = chain.run("Mexican")
print(name)
⟶

→ print(chain.memory.buffer)

⟶ # we see the prev memory stored

Issue can be it can remember every input which takes alot of memory space

* Conversation Chain
* Convo buffer memory goes growing endlessly
* Just remember last 5 Convo Chain
Just remember last 10-20 Convo Chain

→ from langchain.chains import ConversationChain

convo = Co_____(llm=OPENAI(t_____))

print(convo.prompt.template)
⟶

→ convo.run("who ~~~~~~~~~~")

→ print(convo.memory.buffer)

→ shows prev memory

* Conversation Buffer Memory Window

→ from ~~~~~~~~~~ ~WindowMemory

memory = C~~~~~~~~~~(k=1)

↓

Remembers 1 prev convo

convo = C~~ Chain(

llm = OpenAI(~~~~~~~~~~),

memory = memory

)

convo.run("~~~~~~~~~~")

Langchain Document loaders

* from langchain.document_loaders import JSONLoader
  ...
  * and more
  these are used to load documents like PDF, docs, txt, etc.

* In Langchain, the Agent can access many dataframes at once

* Use openAI api

* download dataset ('~.csv')

→ url = "http _____ .csv"
df = pd.read_csv(url)
print(df.shape)
df.head()

→ llm = OpenAI()

→ ~~agent~~
agent = create_pandas_dataframe_agent(llm, df, verbose=True, allow_dangerous_code=True)
agent.run("How many rows are there")

→ 891

→ agent.run("How many people have more than 23 age")

→ 468

* Can also work with Multidataframe

→ df1 = df.copy()

→ agent = create_pandas_dataframe_agent(llm, [df, df1], verbose=True, allow_dangerous_code=True)

* now using both dfs

This is called Multidataframes Agent

Langchain with Huggingface

→ os.env ~~~~~ # api key

→ prompt = PromptTemplate(
input_variables = ["product"],
template = "what is a good name for
a company that makes
{product}"
)

→ chain = LLMChain(llm = HuggingFaceHub(repo_id
= 'google~~~~', model
kwargs = {'temp': 0, 'max
length': 64})

→ chain.run("Colourful Socks")

→ 'sock mania'

Approach 2: Download Model Locally (Create pipelines)

→ from langchain.llms import HuggingFacePipeline
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline, AutoModelForSeq2SeqLM

→ model_id = 'go——'

→ Tokenizer = AutoTokenizer.from_pretrained(model_id)

→ model = AutoModelSeq2SeqLM.from_pretrained(model_id, load_in_8bit=True, device_map='auto')

↓ loading model to gpu
make sure you have good gpu or else slow execution

→ pipeline = pipeline("Text2text-generation", model=model, tokenizer=Tokenizer, max_length=128)

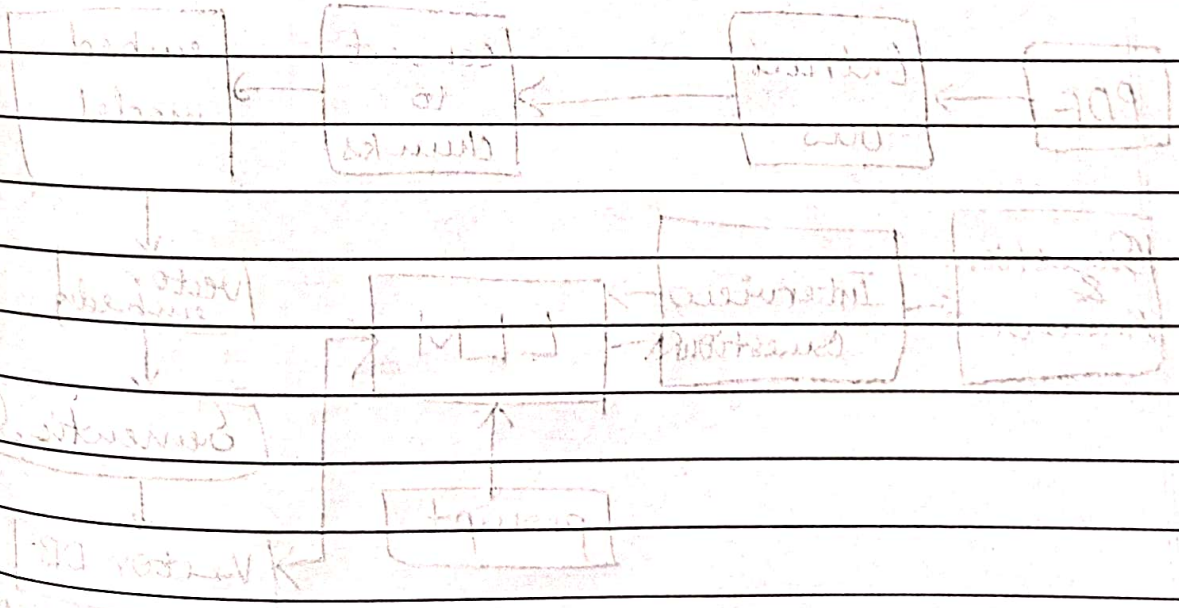
→ local_llm = HuggingFacePipeline(pipeline=pipeline)

→ prompt = PromptTemplate(
input_variables = ["product"],
template = "What is a good name for a company that makes {product}"
)

→ chain = LLMChain(llm=local_llm, prompt=prompt)

→ chain.run("colourful socks")

→ sock mania



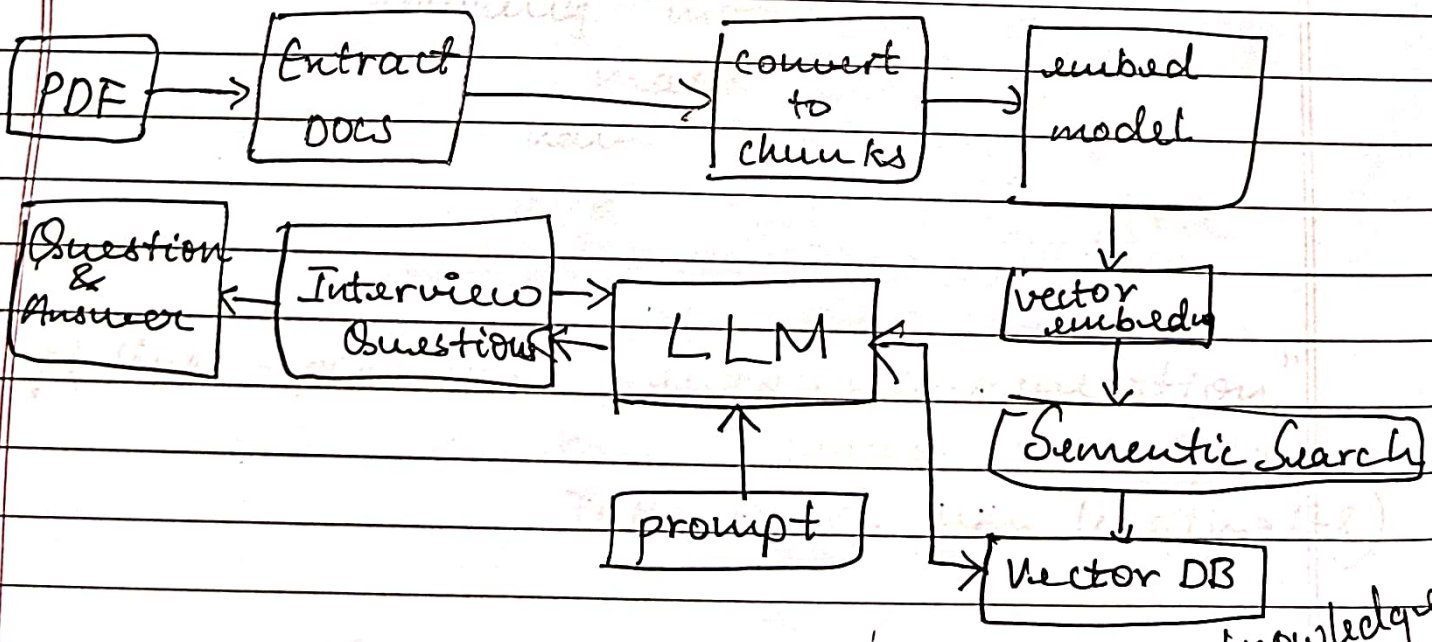
Interview Questions Creator

* If you upload any file of a book or study material the model will give you questions on the study material

Requirements

1. Langchain
2. Open AI – GPT 3.5
3. Vector DB – FAISS (local VDB)
4. Fast-API

Structure

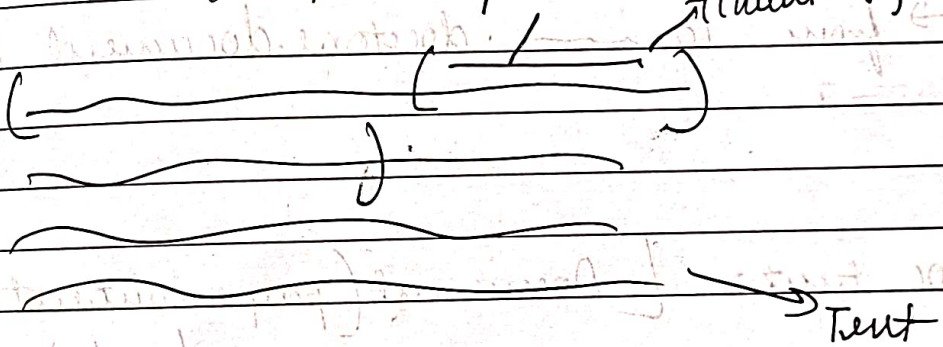


* Check in the video to get and understand the video

* There are different Chunk Methods

1. * Character text Splitter
2. * Recursive Text Splitter
3. * Token Text Splitter

1) We know
   Chunk Size
   Chunk Overlap



so here the sequence is understandable and understandable as the next chunk is having some of same part as previous as there is chunk overlap.

This creates a content

③ Here a LLM is remembering which are in sequence so it is much powerful

* Remember!

before Text splitting the data must also be in document form for that can use

→ from langchain.docstore.document import Document

doc_text = [Document(page_content=t) for t in chunk_text]

Some Open Source LLMs

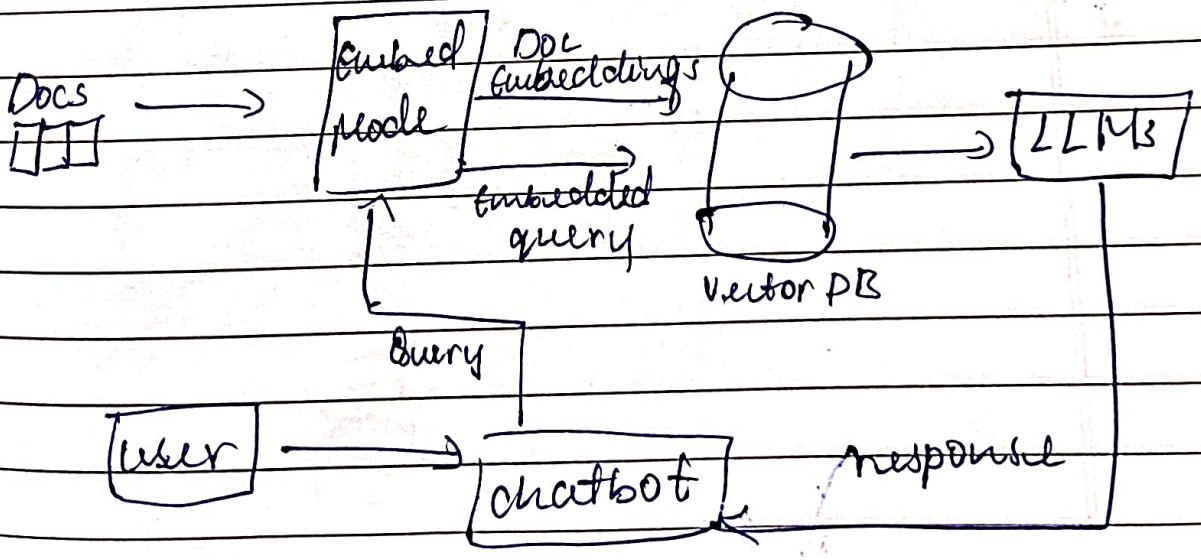
→ Meta Llama 2
→ Google PaLM 2
→ Falcon

Retrieval Augmented GENERATION (RAG)

Limitations of LLMs

~~Limitation~~

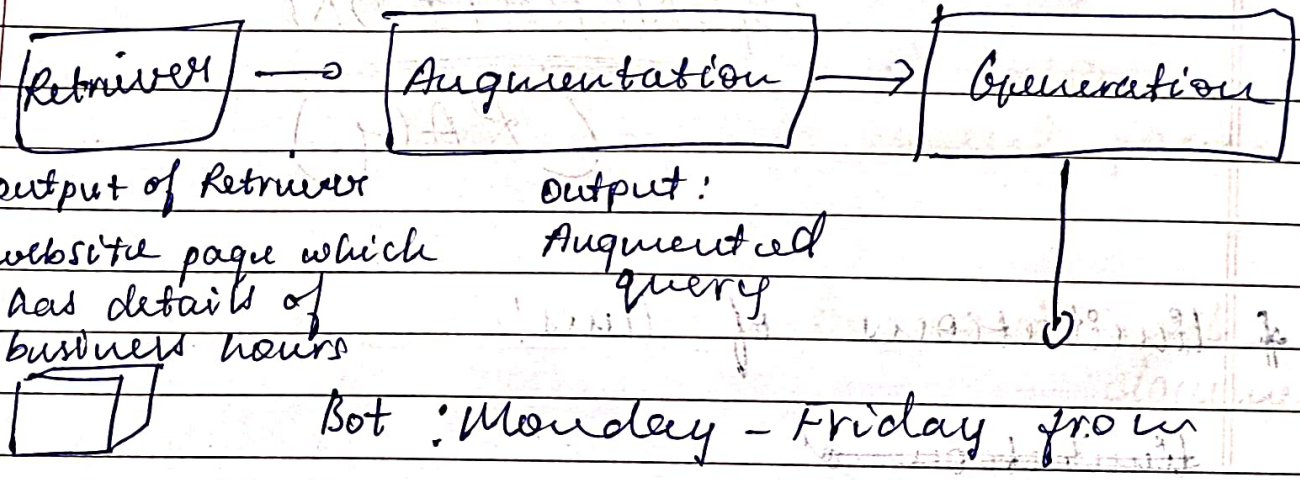
LLMs can't answer correctly to private data
LLMs can't provide the most current info

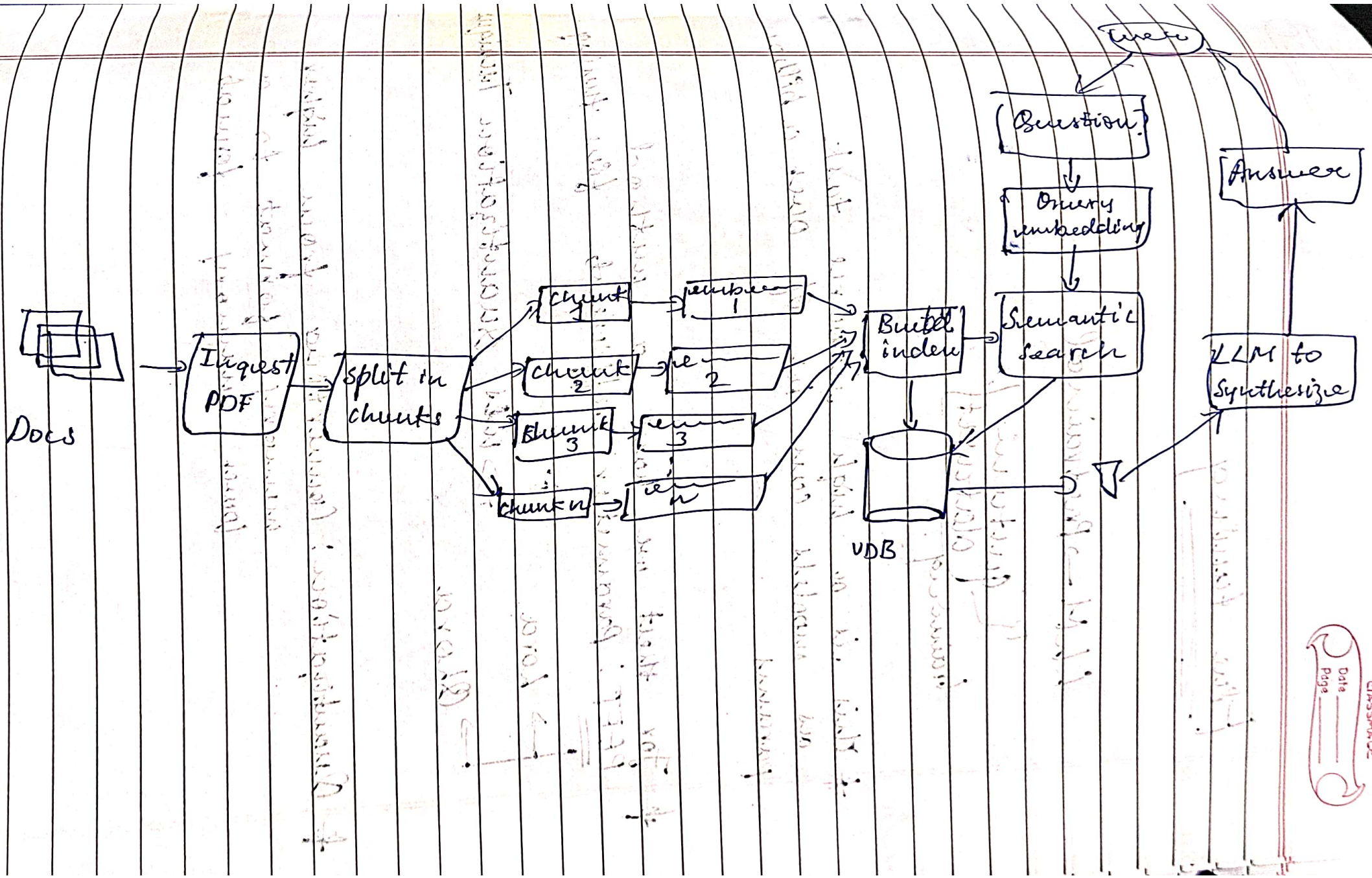


* Rag allows LLM to use external source for better responses

More accurate ans on private data stay updated with new info

* User: What are your business hours?




Docs
Inquest PDF
split in chunks
chunk 1
chunk 2
chunk 3
chunk n
embed 1
e 2
3
n
Build index
VDB
User
Question
Query embedding
Semantic search
LLM to Synthesize
Answer

Fine tuning

LLM → Pretrained

↑ custom dataset

training

* This is a high resource task as models can be of over a billion params

* For that we have a method
PEFT: Parameter efficient fine tuning
  - → Lora
  - → Qlora

  ⇒ uses Quantization Technique

* Quantization: Converting from higher memory format to a lower memory format

So, Lora and QLora help us load these models into Quantized format.

LlamaIndex

An Alternatives framework of Langchain.

→ !pip install -q llama-index

→ !pip install pypdf

→ !pip install docx2txt

→ !pip install google-generativeai

→ !pip install transformers

→ from llama_index import SimpleDirectoryReader
VectorStoreIndex

from llama_index.llms.palm import PaLM
from llama_index import ServiceContext
from llama_index import StorageContext,
load_index_from_storage

import os

loading the file with file path

doc =

→ SimpleDirectoryReader("data").load_data()

→ os.environ['Google_PALM_API_key'] = '————'

→ llm = PaLM()

* In langchain we used to use text splitter but here we are gonna use Service Context

→ Service_Context = Service_Context.from_defaults(llm=llm, chunk_size=800, chunk_overlap=20)

you can also specify which embedding model to use but by default it uses ~~huggingface~~ OpenAI embeddings

So need to setup OpenAI API key or can use hugging face embedding

→ from langchain.embeddings.huggingface import HuggingFaceEmbeddings

→ embed_model = HuggingFaceEmbeddings(model_name="BAAI/bge-base-en")

→ service_context = ServiceContext.from_defaults(embed_model=embed_model)

Now use VectorStore

⇒ index = VectorStoreIndex.from_documents(documents, service_context=service_context)

Storing and loading the Index

→ index.storage_context.persist()

Loading

→ storage_context = StorageContext.from_defaults(persist_dir='./storage')

index = load_index_from_storage(storage_context=storage_context)

* Q/A operation

→ ~~Index~~ Query_engine = index.as_query_engine()

→ responce = query_engine.query("What is llama Index?")

responce

→

This is not understable

* So

→ from Ipython.display import Markdown, display

→ display [Markdown (f"<b>{response} <b>")]

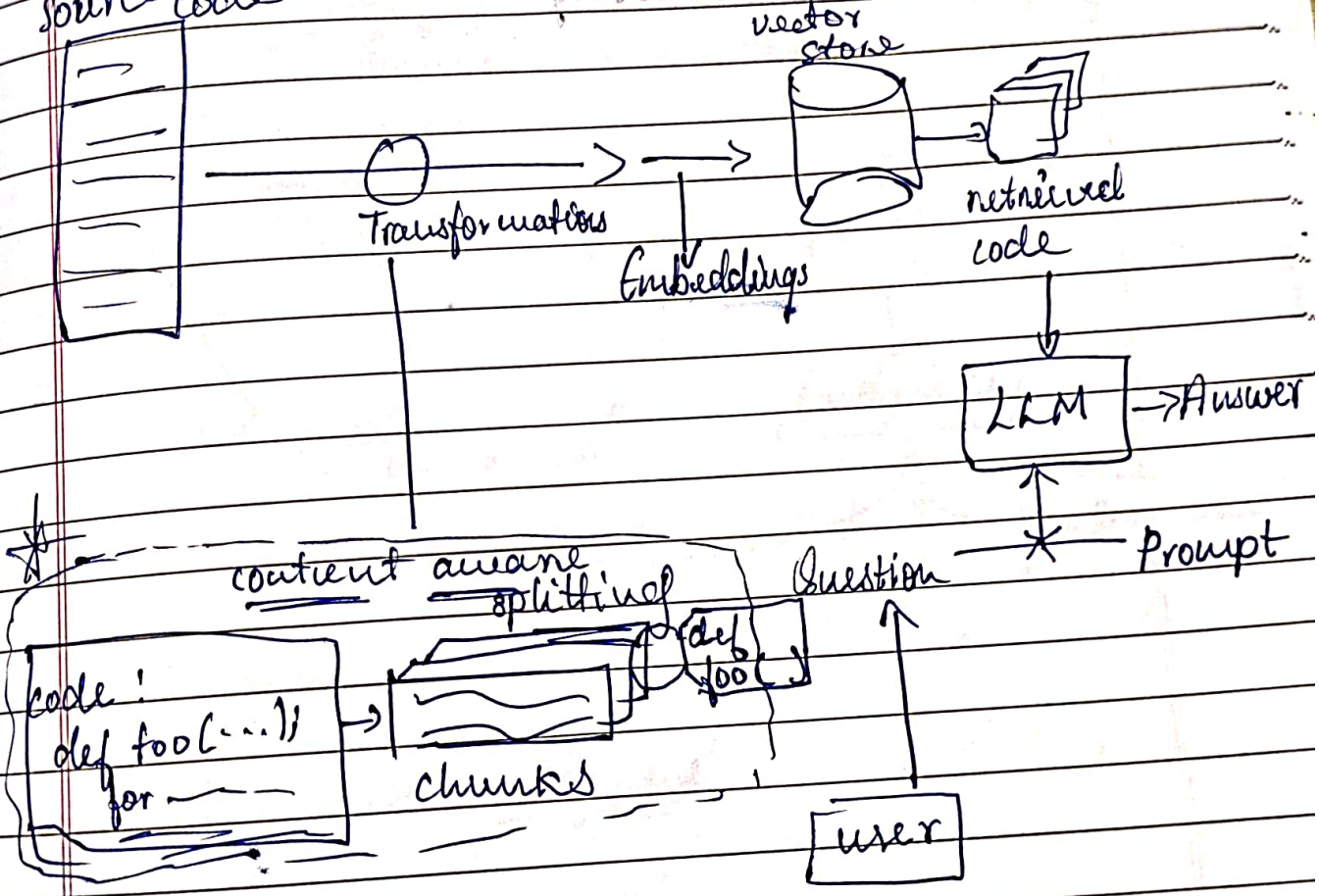
→

* As we see here we didnt use pinecone and all as our knowledge base is the content Index storage of llama

Project :-

Q/A over the Code Base to Understand

how it works



Refer project from video on git hub

Chainlit

* A python ~~like~~ framework where we can create chatgpt like websites easily.