- Locality is given by KNN, whose distance is used to estimate the local density. By comparing the local density of a sample to the local densities of its neighbors, one can identify samples that have a substantially lower density than their neighbors. These are considered outliers.

DOCKER

Containers:

- A way to package application with all the necessary dependencies & configuration.
- Portable artifact; easily share & move this package to any env..
- Makes development & deployment more easy & efficient.

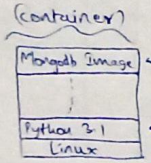
⇒ IMAGE — CONTAINER

- Package/Artifact
- Move/Share this artifact

- Docker image → run → starts the application by creating a container (installs all dependencies & makes an env.)

(container)

or.



(Refer: Docker vs Virtual Machine)

⇒ Docker - Compose: a tool that helps us define & share multi-container applications.

(docker-compose.yml)

⇒ Volumes: preferred mechanism for persisting data generated by & used by docker containers.

PROJECTS

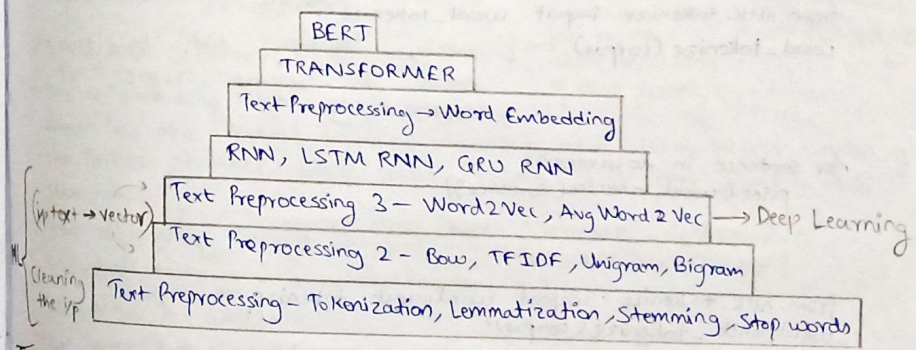
Project 1:

1) Created a repository on Github.
   - In VS code terminal; → conda create -p venv python==3.8 -y (Creates venv folder)
     → conda activate venv/
2) Create .gitignore file.
3) Create setup.py, requirements.txt (pip install -r requirements.txt)
   - setup.py = responsible for creating my ML application as a package. Now we can also install this package in different projects. After making it as a package, we can upload it on Python Pypi and from there anybody can download it.
   - Create src folder & add this file "__init__.py", so that when find_packages() from setup.py searches all the packages needed, it will consider the src folder as a package.
   - Write setup.py
   - In requirements.txt, adding '-e', so that it automatically triggers setup.py
4) Create components folder & create __init__.py file.
   (modules that we are going to specifically use in the project)
5) Creating these files inside components folder: → data_ingestion.py, → data_transformation.py, → model_trainer.py
6) Create pipeline folder. Create these files in this folder: → __init__.py, → train_pipeline.py, → predict_pipeline.py
7) Create logger.py, exception.py, utils.py in src folder. Write own custom exception.py. Write logger.py to log the exceptions which has occurred.
8) Create notebook folder & refer the dataset & both the .ipynb files.
9) Write → data_ingestion.py → data_transformation.py → model_trainer.py
10) Write app.py & predict_pipeline.py
11) Deploying on AWS, Azure.

* Refer: MLFlow, BentoML, DagsHub !

NLP FOR MACHINE LEARNING

Overview:



Tokenization:

- CORPUS = Paragraph
- DOCUMENTS = Sentences
- VOCABULARY = Unique Words
- WORDS = words

Tokenization: process wherein we convert paragraphs into sentences (or sentences into words). (aka Tokens)

Ex: "My name is Ankit. I am a Krishna devotee as well as billionaire by money by age 25." → (CORPUS)

(tokenization) ↓ Converting to tokens → (here: sentences)

1) My name is Ankit.
2) I am a Krishna devotee as well as billionaire by money by age 25.

(tokenization) ↓ converting to words → (tokens)

(. . . . .)

Unique Words = Vocabulary = 16

Ex:

```
!pip install nltk
corpus = """Currently learning NLP to progress. Watch to become
            expert's in NLP."""
print(corpus)
import nltk
nltk.download('punkt')
#Tokenization {Paragraph → words | Sentence → words}
from nltk.tokenize import sent_tokenize →(Algo. 1)
documents = sent_tokenize(corpus)
type(documents) ——→ (list)
for sentence in documents:
    print(sentence)
```

→ Currently learning NLP to progress.
Watch to become expert's in NLP.

```
#Tokenization
from nltk.tokenize import word_tokenize →(Algo. 2)
word_tokenize(corpus) → ['Currently', 'learning', 'NLP', 'to', 'progress', '.',
                         'Watch', 'to', 'become', 'expert', "'s", 'in',
                         'NLP', '.']

for sentence in documents:
    print(word_tokenize(sentence))
```

→ ['Currently', 'learning', 'NLP', 'to', 'progress', '.']
['Watch', 'to', 'become', 'expert', "'s", 'in', 'NLP', '.']

```
from nltk.tokenize import wordpunct_tokenize →(Algo. 3)
wordpunct_tokenize(corpus)
```

→ [, 'become', 'expert', "'", 's', 'in', 'NLP', '.']
(2D Array)

```
from nltk.tokenize import TreebankWordTokenizer →(Algo. 4)
tokenizer = TreebankWordTokenizer()
tokenizer.tokenize(corpus)
```

→ [. . ., 'to', 'progress.', 'Watch', . . ., 'expert', "'", 's', 'in',
'NLP', '.']

- Spacy:
  - Need to download packages related to English text.
  - Provides most efficient NLP algo. for a given task.
  - New library & popular amongst developers.

- NLTK:
  - Natural Language Toolkit
  - It's a string processing library & it provides access to many algorithms.
  - Researchers prefer to use NLTK in their work.

Stemming:

It's the process of reducing a word to its word stem that affixes to suffixes & prefixes or to the roots of words known as a 'lemma'. Stemming is imp. in NLU & NLP.

Ex:

```
words = ["eating", "eaten", "history", "congratulations", "finally", "fairly"]
#PorterStemmer → (Algo. 1)
from nltk.stem import PorterStemmer
stemming = PorterStemmer()
for word in words:
    print(stemming.stem(word))
```

op: eat
eat
congratul
final
fairli

#RegexpStemmer Class → (Algo. 2)

#NLTK has (↓) class with the help of which we can easily implement Regular Expression Stemmer Algorithms. It basically takes a single regular expression & removes any prefix or suffix that matches the expression. Let us see an example:

```
from nltk.stem import RegexpStemmer
reg_stemmer = RegexpStemmer('ing$|s$|e$|able$', min=4)
reg_stemmer.stem('eating') → eat
reg_stemmer.stem('ingeating') → ingeat
```

min. len. of string to do stemming

if last letters = ing, s, e, able
Removing '$' will remove whole occurence of that word/character; it will remove it.

#Snowball Stemmer → (Algo. 3)

#It's a stemming algo. which is also known as the Porter2 stemming algorithm as it is a better version of the Porter Stemmer since some issues of it were fixed in this stemmer.

```
from nltk.stem import SnowballStemmer → supports many languages
snowballstemmer = SnowballStemmer('english')
for word in words:
    print(snowballstemmer.stem(word))
```

op: eat
eat
histori
congratul
final
fair

Disadvantage: ⇒ Words are not coming in correct form. The meaning of word is basically changing.

⇒ In order to prevent this, we use something called as 'Lemmatizer'.

Lemmatizer:

Lemmatization technique is like stemming. The o/p we will get after lemmatization is called 'lemma' which is a root word rather than root stem, the o/p of stemming. After lemmatization, we will be getting a valid word that means the same thing.

Ex: # WordNet Lemmatizer

#NLTK provides WordNetLemmatizer class which is a thin wrapper around the wordnet corpus. This class uses morphy() function to the WordNet CorpusReader class to find a lemma.

Use/usecase: QnA, chatbots, text summarization.

```
import nltk
nltk.download('wordnet')
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
'''
pos = Pos Tag → Noun - n (means 'noun' denoted as 'n')
              → Verb - v
              → adjective - a
              → adverb - r
'''
lemmatizer.lemmatize("going", pos='v') → o/p: go

for word in words:
    print(lemmatizer.lemmatize(word, pos='v'))
```

eat
eat
history
congratulations
finally
fairly

Stopwords:

Stopwords are crucial in NLP to remove common words that add little semantic value, improving the efficiency & effectiveness of text analysis. (Refer Notebook for output)

Ex:
```
paragraph = """ I have three visions for India. (...) """
from nltk.corpus import stopwords
import nltk
nltk.download('stopwords')
stopwords.words('english')  } (available in diff. languages)
stopwords.words('arabic')   }
from nltk.stem import SnowballStemmer
snowballstemmer = SnowballStemmer('english')
# Apply Stopwords → filter → apply snowball stemming
for i in range(len(sentences)):
    words = nltk.word_tokenize(sentences[i])
    words = [snowballstemmer.stem(word) for word in words if word not in
                                        set(stopwords.words('english'))]
    sentences[i] = ' '.join(words) # converting all the list of words into
                                                          sentences
sentences # snowball also ensures that all the words are in small case & converts any, if found in uppercase.
```

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
for i in range(len(sentences)):
    words = nltk.word_tokenize(sentences[i])
    words = [lemmatizer.lemmatize(word.lower(), pos='v') for word in words
                                    if word not in set(stopwords.words('english'))]
    sentences[i] = ' '.join(words)
sentences
```

Parts of Speech Tags:

Ex:
- CC → coordinating conjunction
- MD → modal (ex: could, will)
- NN → noun, singular (ex: desk)
- NNPS → proper noun, plural (ex: Indians)
- PRP → personal pronoun (ex: I, he, she)
- RBR → adverb, comparative (ex: better)
- RBS → adverb, superlative (ex: best)

(Refer Google/Notebook for all POS Tags)

Ex:
```
import nltk
from nltk.corpus import stopwords
print(nltk.pos_tag("Taj Mahal is a beautiful Monument".split()))
```
takes i/p in the form of list of words

o/p: [('Taj', 'NNP'), ('Mahal', 'NNP'), ('is', 'VBZ'), ('a', 'DT'), ('beautiful', 'JJ'), ('Monument', 'NN')]

(determiner)

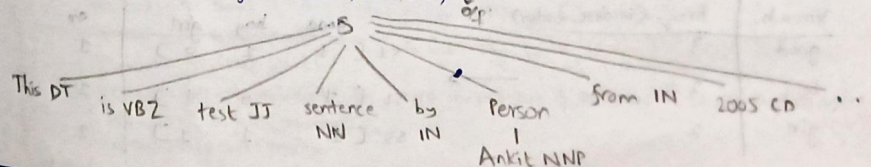
NNP = proper noun
VBZ = verb, 3rd person singular present tense
JJ = adjective

```
# If we have paragraph & we want to find POS Tag of each word
for i in range(len(sentences)):
    words = nltk.word_tokenize(sentences[i])
    words = [word for word in words if word not in set(stopwords.words
                                                    ('english'))]
    pos_tag = nltk.pos_tag(words)
    print(pos_tag)
```

Named Entity Recognition:

Types of tags (ex: person, place/location, date, time, money, percent, ...) (ex: 1 million dollar)

Ex:
```
sentence = "This is test sentence by Ankit from 2005"
import nltk
words = nltk.word_tokenize(sentence)
tag_elements = nltk.pos_tag(words)
nltk.download('maxent_ne_chunker')
nltk.download('words')
nltk.ne_chunk(tag_elements).draw()
```



Flow:

ex: We are doing "sentiment analysis" ;

Dataset → Text preprocessing 1 → Text preprocessing 2 → Text → Vectors → ML/DL Algo.-s

- Text preprocessing 1: 1) Tokenization 2) Lowercase the words 3) Regular expression
- Text preprocessing 2: 1) Stemming 2) Lemmatization 3) Stopwords
- Text → Vectors: 1) One Hot Encoded 2) Bag of Words (BOW) 3) TF-IDF 4) Word2Vec 5) AvgWord2Vec

1) One Hot Encoding: → (ex: sentiment analysis) → aim = convert word to vector

ex:

Text	o/p
D1: The food is good	1
D2: The food is bad	0
D3: Pizza is Amazing	1

⇒ Vocabulary size = 7

↓

The food is good bad Pizza Amazing

D1: [[1 0 0 0 0 0 0], [0 1 0 0 0 0 0], [0 0 1 0 0 0 0], [0 0 0 1 0 0 0]] — 4×7

D2: [[1 0 0 0 0 0 0], [0 1 0 0 0 0 0], [0 0 1 0 0 0 0], [0 0 0 0 1 0 0]] — 4×7

D3: [[0 0 0 0 0 1 0], [0 0 1 0 0 0 0], [0 0 0 0 0 0 1]] — 3×7

- Advantages:
  - easy to implement with python [sklearn → OneHotEncoder , pd.get_dummies()]
- Disadvantages:
  - Sparse matrix → leads to overfitting
  - For ML Algo.-s, we need fixed size data (here we have 3×7, 4×7)
  - No semantic meaning is getting captured.
  - Out of Vocabulary (OOV) (i.e. if in test data, a word comes which is not in vocab)

2) Bag of Words: → (ex: sentiment analysis)

ex:

Text	o/p
He is a good boy	1
She is a good girl	1
Boy and girl are good	1

⇒ Lowercase all the words ⇒ apply stopwords

S1 → good boy
S2 → good girl
S3 → boy girl good

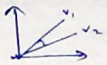
Vocab.	Freq. (in desc. order)
good	3
boy	2
girl	2

	good	boy	girl	o/p
S1	1	1	0	1
S2	1	0	1	1
S3	1	1	1	1

- Binary BOW (1 and 0) ↓ (Prev. ex.) and BOW (count will get updated based on freq.)

- Advantages:
  - simple & intuitive
  - fixed size i/p (it will help ML Algo.-s)
- Disadvantages:
  - Sparse matrix/array → overfitting
  - Out of Vocabulary (OOV)
  - Order of the word is getting changed (based on freq. of word → refer ex.)
  - semantic meaning is still not captured.

ex: The food is good → [1 1 1 0 1] → v1
The food is not good → [1 1 1 1 1] → v2
} cosine similarity →



o/p ∴ similar BUT in actual they're complete opposite

3) N-grams:

ex: S1 → The food is good
S2 → The food is not good
} after removing stopwords →

	food	not	good
S1	1	0	1
S2	1	1	1

→ (unigram)

(BIGRAM)

	food	not	good	food good	food not	not good
S1	1	0	1	1	0	0
S2	1	1	1	0	1	1

Now our model can clearly distinguish btw both sentences. (As there are many mismatches)

n-grams = (1,1) → unigrams
= (1,2) → unigram, bigram → (combs. of 2 words)
= (1,3) → unigram, bigram, trigram → (comb. of 3 words)
= (2,3) → bigram, trigram

Ex: (Bag of Words + N-grams) → Refer notebook for more...

```
messages = pd.read_csv('<location/>', sep='\t', names=["label", "message"])   # sep = separator
messages
# Data Cleaning & Preprocessing
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
ps = PorterStemmer()
corpus = []
for i in range(0, len(messages)):
    review = re.sub('[^a-zA-Z]', ' ', messages['message'][i])
    review = review.lower()
    review = review.split()
    review = [ps.stem(word) for word in review if not word in stopwords.words('english')]
    review = ' '.join(review)
    corpus.append(review)
```

(14) corpus

#Create BOW

```
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer(max_features=100, binary=True)   # for binary BOW
x = cv.fit_transform(corpus).toarray()                # (selecting top 100 vocab. wrt frequency)
x.shape
x
```

#N-grams

```
cv.vocabulary_   # Gives top 100 words → (unigrams only)
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer(max_features=100, binary=True, ngram_range=(2,3))
x = cv.fit_transform(corpus).toarray()
cv.vocabulary_   → gives top 100 words (Bigram + Trigram)
x
```

4) TF-IDF: (Term Frequency - Inverse Document Frequency)

ex: S1 → good boy
S2 → good girl
S3 → boy girl good
(from prev. ex. after preprocessing)

$$TF = \frac{\text{No. of repetition of words in sentence}}{\text{No. of words in sentence}}$$

$$IDF = \log_e\left(\frac{\text{No. of sentences}}{\text{No. of sentences containing the word}}\right)$$

TF

	S1	S2	S3
good	1/2	1/2	1/3
boy	1/2	0	1/3
girl	0	1/2	1/3

* IDF

Words	IDF
good	$\log_e(3/3)=0$
boy	$\log_e(3/2)$
girl	$\log_e(3/2)$

Note: If a word is present in every sentence → Less priority

=

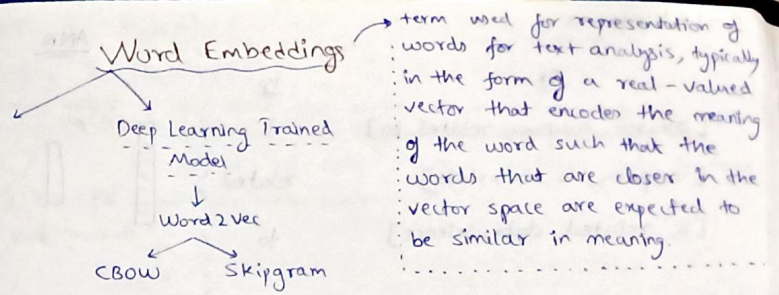
	[good	boy	girl]
S1	0	$\frac{1}{2}\times\log_e(3/2)$	0
S2	0	0	$\frac{1}{2}\times\log_e(3/2)$
S3	0	$\frac{1}{3}\log_e(3/2)$	$\frac{1}{3}\log_e(3/2)$

o/p → (given)

- Advantages:
  - Intuitive
  - Fixed Size → Vocab Size
  - Word Importance is getting captured
- Disadvantages:
  - Sparsity still exists
  - OOV

Ex. (14) → Lemmatize instead of stemming

```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf = TfidfVectorizer(max_features=100)
x = tfidf.fit_transform(corpus).toarray()   → top 100 vocabs wrt frequency
x
#N-Grams
tfidf = TfidfVectorizer(max_features=100, ngram_range=(2,2))
x = tfidf.fit_transform(corpus).toarray()
tfidf.vocabulary_
```



5) Word2Vec:

- This algo. uses a neural network model to learn word associations from a large corpus of text. As the name implies, word2vec represents each distinct word with a particular list of numbers called a vector.
- Each word in vocabulary is converted to 'Feature representation', means each word is converted into vector, based on some features.

ex:

	Boy	apple	mango
Gender	-1	0.01	0.05
Food	0.01	-0.02	0.02
Age	0.03	0.09	0.01
⋮	⋮	⋮	⋮

(300 dimensions) → used by 'Google'

(is a unsupervised ML Algo.)

Word2Vec
- CBOW (Continuous Bag of Words)
- Skipgram

5.1) CBOW (Continuous Bag of Words)

ex: corpus = dataset → [iNeuron Company is related to data science] (Vocab.)

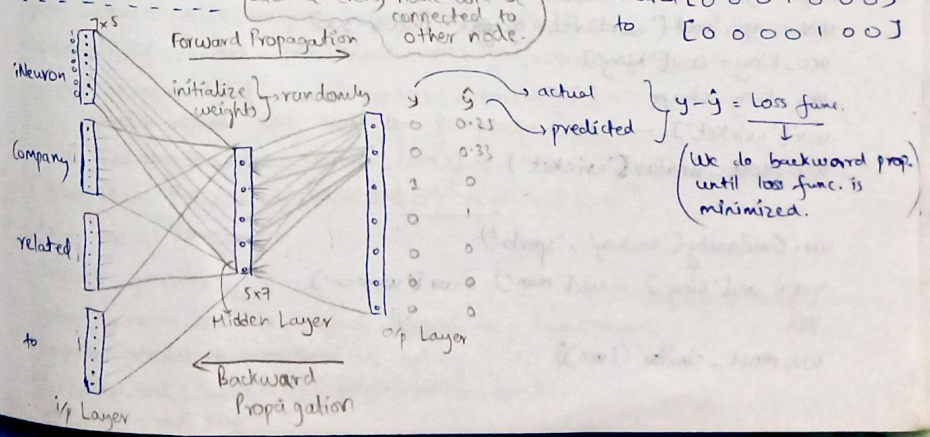
Let window size = 5 & now taking it's 'middle' word.

i/p	o/p
[iNeuron, company, related, to]	is
[company, is, to, data]	related
[is, related, data, science]	to

(While sending it to process in ML Algo., we convert it into OHE

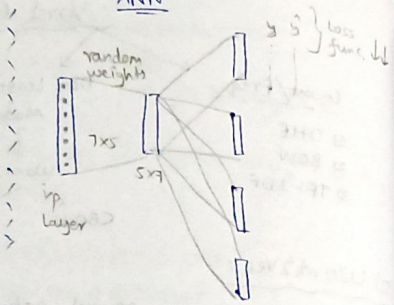
ex: iNeuron [1 0 0 0 0 0 0]
company [0 1 0 0 0 0 0]
related [0 0 0 1 0 0 0]
to [0 0 0 0 1 0 0]

Fully Connected NN: → (Each & every node will be connected to other node.)



5.2) Skipgram

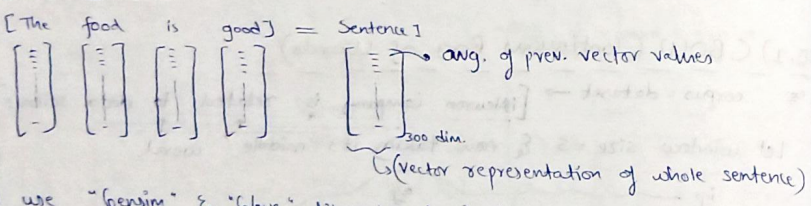
i/p	o/p
[iNeuron, company, related, to]	is
[company, is, to, data]	related
[is, related, data, science]	to



* We apply 'softmax' func. to get 'ŷ'.

- Improving: → Increase the training data.
  → Increase the window size = vector dimension is also increasing.
- When to apply: → Small dataset = CBOW
  → Large dataset = Skipgram
- Advantages of Word2Vec:
  - Dense Matrix
  - Semantic info. is getting captured.
  - OOV is also solved.
  - Vocab. Size → fixed set of dimension vectors
    ↳ Google word2vec (300 dimensions)

6) Average Word2Vec



↳ (Vector representation of whole sentence)

We will use "Gensim" & "Glove" libraries to implement.

Ex: (Word2Vec)

```
!pip install gensim
import gensim
from gensim.models import Word2Vec, KeyedVectors
import gensim.downloader as api
wv = api.load('word2vec-google-news-300')
vec_king = wv['king']
vec_king.shape → (300,)
wv['cricket'] → gives values of all those 300 vector values
wv.most_similar('cricket') → [(most_similar_words, % matching)]
                              (sp format)
wv.similarity('hockey', 'sports')
vec = wv['king'] - wv['man'] + wv['woman']
vec
wv.most_similar([vec])
```

Ex: (BOW, TFIDF ML Algo.'s)

Flow →
1) Preprocessing & Cleaning
2) Train Test Split : We want our model to have no info. on test-data, so before changing sentences into vector, we split it hence avoiding data leakage.
3) BOW & TF-IDF {Sentences → Vectors}
4) Train Our models.

(14)

```
#Create BOW
y = pd.get_dummies(messages['label'])
y = y.iloc[:,0].values  # Just taking 'ham' values
```

(0) → `train_test_split(corpus, y, test_size=0.25, random_state=42)`

```
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer(max_features=2500, ngram_range=(1,2))
len(x_train), len(y_train)
#Independent Features
x_train = cv.fit_transform(x_train).toarray()
x_test = cv.transform(x_test).toarray()
x_train
cv.vocabulary_
from sklearn.naive_bayes import MultinomialNB → Multi. Naive Bayes
spam_detect_model = MultinomialNB().fit(x_train, y_train)
y_pred = spam_detect_model.predict(x_test)
```

(6)

```
#Creating TF-IDF Model
```

(→)

```
from sklearn.feature_extraction.text import TfidfVectorizer
tv = TfidfVectorizer(max_features=2500, ngram_range=(1,2))
x_train = tv.fit_transform(x_train).toarray()
x_test = tv.transform(x_test).toarray()
x_train
tv.vocabulary_
from sklearn.naive_bayes import MultinomialNB
spam_tfidf_model = MultinomialNB().fit(x_train, y_train)
y_pred = spam_tfidf_model.predict(x_test)
```

(6)

Ex: (Spam Ham Project = Word2Vec, AvgWord2Vec)

```
!pip install gensim
import gensim
from gensim.models import Word2Vec, KeyedVectors
import gensim.downloader as api
wv = api.load('word2vec-google-news-300')
vec_king = wv['king']
```

vec_king

(14) → Lemmatize instead of stemming

```
[[i,j,k] for i,j,k in zip(list(map(len, corpus)), corpus, messages['message'])
                                                    if i<1]
```

Substitute for (14)

```
from nltk import sent_tokenize
from gensim.utils import simple_preprocess
words = []
for sent in corpus:
    sent_token = sent_tokenize(sent)
    for sent in sent_token:
        words.append(simple_preprocess(sent))
words
```

```
# Training Word2Vec from Scratch
model = gensim.models.Word2Vec(words)
model.wv.index_to_key # To get all the vocabulary
model.corpus_count # Vocabulary Size
model.epochs # Higher epochs = better performance of model
model.wv.similar_by_word('good')
model.wv['good'].shape
words[0]
def avg_word2vec(doc):
    return np.mean([model.wv[word] for word in doc if word in
                    model.wv.index_to_key], axis=0)
```

→ word for word in doc if word in model.wv.index_to_key

→ Remove out of vocabulary words!

```
!pip install tqdm
from tqdm import tqdm
import numpy as np
x = []
for i in tqdm(range(len(words))):
    x.append(avg_word2vec(words[i]))
```

len(x) → Refer why 3 values are missing!

```
# Independent Features
x_new = np.array(x)
messages.shape
x[1] # Vectors of 2nd sentence in paragraph
x_new.shape
x_new[0].shape
y = messages['label'][list(map(lambda x: len(x)>0, corpus))]
y = pd.get_dummies(y['label'])
y = y.iloc[:,0].values
y.shape
x[0].reshape(1,-1).shape
```

x[0] shape = (100,)
Now to add it as a row, we reshape it to (1,-1) → so new shape = (1, 100)
row, col.s

```
# Final independent features
df = pd.DataFrame()
for i in range(0, len(x)):
    df = df.append(pd.DataFrame(x[i].reshape(1,-1)), ignore_index=True)
df.head()
df['Output'] = y
df.head()
df.dropna(inplace=True)
df.isnull().sum()
# Independent Features
x = df
x.isnull().sum()
y = df['Output']
```

⑥

```
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier()
classifier.fit(x_train, y_train)
y_pred = classifier.predict(x_test)
```

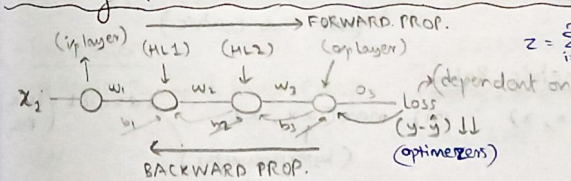
⑥

Ex: (Refer Notebook → kindle Review Sentiment Analysis)

DEEP LEARNING

Refer "Application of Derivatives in Deep Learning Neural Network" → PREV. CHAPTER

Vanishing Gradient Problem & Activation functions:



$z = \sum_{i=1}^{n} w_i x_i + b$; $\sigma(z) = 0$ to 1 (Sigmoid) $= \left(\frac{1}{1+e^{-z}}\right)$

(dependant on o_3)

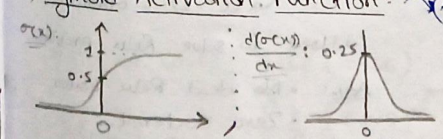
* $0 \leq \sigma(z) \leq 1$
* Derivative $(\sigma(z))$ $\Big\}$ $0 \leq \sigma(z) \leq 0.25$

$W_{new} = W_{old} - \eta\left(\frac{\partial L}{\partial W_{old}}\right)$ → (Small Val.)

$\left\{\frac{1}{\text{Big}}\right\}$

The "Vanishing Gradient" occurs when gradients used for updating weights during training becomes very small. this makes it hard for the network to learn & update its weights effectively.

1) Sigmoid Activation Function: (transforms value btw 0 to 1)

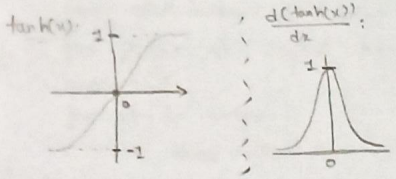


$\sigma(x) = \frac{1}{1+e^{-x}}$

Adv.:
- Binary classification suitable
- Clear prediction (1 or 0)

Disadv.: • Prone to vanishing gradient problem.
• Func. output is not zero centered → (efficient weight updation)
• Mathematical operation are relatively time consuming.

2) Tanh Activation Function:



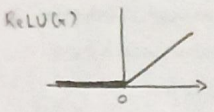
$\frac{d(\tanh(x))}{dx}$:

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

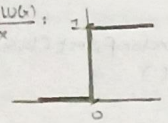
Adv.: • zero centric → weight updation is efficient

Disadv.: • Prone to vanishing gradient prob.
• Time complexity

3) Relu Activation Function:
(Rectified Linear Unit)



$\frac{dReLU(x)}{dx}$:



$ReLU = \max(0, x)$

- If derivative of relu either → o/p = 1 (wt. updation will happen) or → o/p = 0 (dead neuron)
↓
$(w_{new} \approx w_{old})$

Adv.: • Solving Vanishing Gradient problem
• $\max(0, x)$ → calculation is superfast. The Relu func. has a linear relationship.
• It is much faster than sigmoid or Tanh

Disadv.: • Dead Neuron
• Relu func. o/p → either '0' or 'x'; x ∈ +ve number. (Not zero centric)

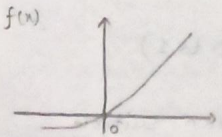
4) Leaky Relu & Parametric Relu:

Here: $f(x) = \max(0.01x, x)$ (or) $\max(\alpha x, x)$ (hyperparameter) → $\alpha = 0.01$ → (Leaky Relu)

Adv.: • (All advantages of Relu)
• Removes Dead Relu problem. (or) (Dead Neuron)

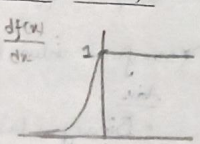
Disadv.: • It is not zero-centric.

5) ELU (Exponential Linear Units)



(F/w prop.)

$\frac{df(x)}{dx}$:



(B/w prop.)

$$f(x) = \begin{cases} x & ; \text{if } x > 0 \\ \alpha(e^x - 1) & ; \text{otherwise} \end{cases}$$

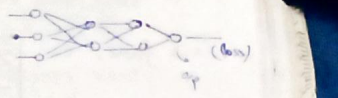
→ Used to solve Relu problem.

Adv.: • No dead Relu issues.
• Zero centered.

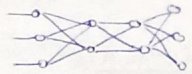
Disadv.: • Slightly more computationally intensive.

6) Softmax Activation Function:

- Sigmoid activation → when o/p = binary classification ⇒



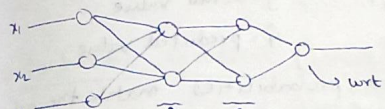
- Softmax act. func. → when o/p = multiclass classification



$\frac{e^{y_i}}{\sum_{k=0}^{n} e^{y_k}}$; $y_i = O * w + b$

n = no. of output nodes

Which Activation Func. to use when?



w.r.t o/p → Sigmoid = binary classification
→ Softmax = multiclass classification.

We use: (Relu (or) it's variants)

Loss func. & Cost func.:

ex	x_1	x_2	x_3	o/p
	-	-	-	0
	-	-	-	1
	-	-	-	0
	-	-	-	1

Loss func.: $MSE = (y - \hat{y})^2$ → We take every single datapoint & do f/w - b/w propagation.

Cost func.: We take all the datapoints at once & do f/w - b/w propagation.

$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$

ANN → Classification
→ Regression

- Regression:

1) Mean Squared Error (MSE):
• Loss func. = $(y - \hat{y})^2$
• Cost func. = $\frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$

Adv.:
- MSE is differentiable
- It has 1 local/global minima (Gradient descent)
- It converges faster.

Disadv.:
- Not robust to outliers

2) Mean Absolute Error (MAE):
• Loss func. = $|y - \hat{y}|$
• Cost func. = $\frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|$

Adv.: - Robust to outliers

Disadv.: - convergence usually takes time in MAE

3) Huber Loss: (= MAE + MSE)

• Cost func. = $\begin{cases} \frac{1}{n} \sum_{i=1}^{n} (y - \hat{y})^2 & ; \text{if } |y - \hat{y}| \leq \delta \text{ (no outlier)} \\ \delta |y - \hat{y}| - \frac{1}{2}\delta^2 & ; \text{otherwise} \end{cases}$ → (hyperparameter)

4) RMSE (Root Mean Squared Error):

- Cost func. = $\sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{N}}$
- Adv. - Gives more importance to big errors, helping to reduce large mistakes
- Disadv.: - Not robust to outliers

- Classification:

Cross Entropy →
- Binary Cross Entropy {Binary}
- Categorical Cross Entropy {Multiclass}
- Sparse categorical cross entropy {Multiclass}

1) Binary Cross Entropy:

- loss = $-y * \log(\hat{y}) - (1-y) * \log(1-\hat{y})$; y = actual value, $\hat{y}$ = predicted value
  (log loss) → same used in = (Logistic Regression)
- Evaluates how well the predicted probabilities match the actual binary labels.
- Used in binary classification
- $\hat{y} = \frac{1}{1+e^{-z}}$ } sigmoid Act. func

2) Categorical Cross Entropy:

- Loss func. = $L(x_i, y_i) = -\sum_{j=1}^{c} y_{ij} * \ln(\hat{y}_{ij})$; c = no. of categories
- Converts o/p into OHE ~ ex:

o/p	Good	Bad	Neutral
Good	1	0	0
Bad	0	1	0
Neutral	0	0	1

- o/p ~ $\hat{y}_{ij}$ = probabilities (ex: [0.2, 0.3, 0.5])

∴ This also gives the prob. of other categories.

3) Sparse Categorical Cross Entropy:

Disadv.: - Loosing info. about the prob. of other category.

- Final o/p = index of category (ex: [0.2, 0.3, 0.5] → o/p = 2nd index)

*Right Combination:

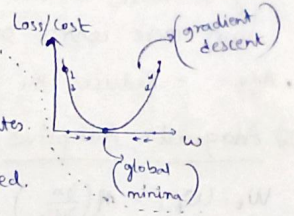
Hidden Layers	o/p Layer (Activation applied)	Problem Statement	Loss Function
1) ReLU or its variants	Sigmoid	Binary Classification	Binary Cross Entropy
2) ReLU or its variants	Softmax	Multi class	Categorical / Sparse CE
3) ReLU or its variants	Linear	Regression	MSE, MAE, Huber Loss, RMSE

Optimizers: (responsible for reducing the loss func. in the back prop.) (changes wt basically)

1) Gradient Descent Optimizer:

weight updation formula ⇒ $W_{new} = W_{old} - \eta\left(\frac{\partial h}{\partial W_{old}}\right)$ ← Learning rate

- Epoch = 1 f/w prop. with ALL datapoints & 1 b/w prop. {1 epoch = 1 iteration}
- This optimizer takes all datapoints at once & calculates.
- 'Epochs' done until loss func. is not reduced.



- Adv: - Convergence will happen
- Disadv: - Huge resource (RAM, GPU) needed.

2) Stochastic Gradient Descent (SGD):

- Takes 1 record at a time ← but (Similar to gradient descent opt.)
- Adv.: - solves resource issue
- Disadv.: - Time complexity ↑
  - Convergence will also take more time.
  - Noise gets introduced.

(path of convergence is not very smooth)

// Since we are solving 1 datapoint at a time, weight updation doesn't happen efficiently & we get noise (zig-zag type).

3) Mini Batch SGD:

(1000 datapoints ← ex:)

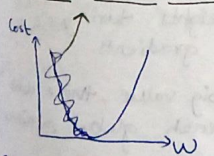
- 3 terms: epoch, iteration, batch_size
- No. of itr's = $\frac{\text{total datapoints}}{\text{batch\_size}}$ = let 100

1 epoch { Itr. 1 (1 datapoint → wt → $\hat{y}$), Itr. 2, ..., Itr 1000 }

- 1 epoch = 100 itr.
- Adv.: - Convergence speed will increase
  - Noise will be less when compared to SGD.
  - Efficient Resource Usage
- Disadv.: - Noise still exists

4) SGD with Momentum:

(smoothening) (above ones also consider this (bias updation))



$W_t = W_{t-1} - \eta \frac{\partial h}{\partial W_{t-1}}$; W_{new} ← W_{old}

, $b_{new} = b_{old} - \eta\left(\frac{\partial h}{\partial b_{old}}\right)$

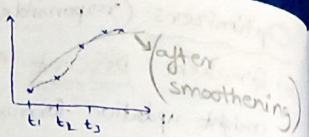
Smoothening happens with the help of "Exponential Weight Average (EWA)"

ex: Time Series ⇒ Time = t_1 t_2 t_3 ... t_n ; Value at time t_1 ⇒ $V_{t_1} = a_1$
Value = a_1 a_2 a_3 ... a_n

$V_{t_1} = \beta * V_{t_1} + (1-\beta) * a_2$ ∴ if ↝

(controls smoothening func.)

(if 'β' = more, we are saying 'a,' should have more control over the next value)

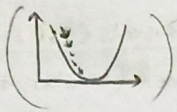


• Adv.: - Reduces the noise - Quick convergence

5) Adagrad: Adaptive Gradient Descent

$W_t = W_{t-1} - \eta\left(\frac{\partial h}{\partial w_{t-1}}\right)$ (used)

Prev. Learning rate = fixed (but here, we have dynamic learning rate)



i.e. As the convergence happens, the learning rate should change.

• $\eta' = \frac{\eta}{\sqrt{\alpha_t + \varepsilon}}$ (small value) (epsilon)

• $\alpha_t = \sum_{i=1}^{t}\left(\frac{\partial h}{\partial w_t}\right)^2$ (derivative of loss wrt W_t)

(traversing btw hidden layers)

• Disadv. & - η' → possibility to become a very small value ≈0

∴ (wt updation not occurring) → $\{w_t \approx w_{t-1}$

6) Adadelta & RMSPROP:

$\eta' = \frac{\eta}{\sqrt{S_{dw_t} + \varepsilon}}$; $S_{dw_t} = \beta * S_{dw_{t-1}} + (1-\beta)\left(\frac{\partial h}{\partial w_{t-1}}\right)^2$ ∴ (Basically smoothening 'α_t')

$\left(w_t = w_{t-1} - \eta'\left(\frac{\partial h}{\partial w_{t-1}}\right)\right)$

(• Dynamic LR + Smoothening EWA)

7) Adam Optimizer: {Best Optimizer}

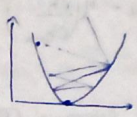
• SGD with momentum + RMSPROP (Dynamic LR + Smoothening)

↳ (All these updates happen in 'weight updation formula' & 'bias updation formula')

Exploding Gradient Problem:

• Occurs when gradients become excessively large during training the DNN. (↑ slope of a func.)

• Happens due to repeated multiplication of large weights during backpropagation, leading to exponentially increasing gradients.



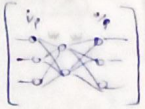
(since wt. upd. occurring by a very big value, there will be a scenario where we won't reach global minima.)

↓

(Must ensure that weights are initialised properly, else, if initialised with very high value → above prob. can come)

Weight Initialising Techniques:

Points to be remembered: - Weights should be small
- Weights should not be same
- Weights should have good variance.



1) Uniform Distribution:

• We initialize the weights based on some uniform distribution.

• $W_{ij} \approx$ uniform distribution $\left[\frac{-1}{\sqrt{input}}, \frac{1}{\sqrt{input}}\right]$ (always give range)

2) Xavier / Glorot Initialization:

• Xavier Uniform Initialization: $W_{ij} \approx$ uniform distribution $\left[\frac{-\sqrt{6}}{\sqrt{input + output}}, \frac{\sqrt{6}}{\sqrt{input + output}}\right]$

• Xavier Normal Initialization: $W_{ij} \approx N(0, \sigma)$; $\sigma = \sqrt{\frac{2}{(input + output)}}$

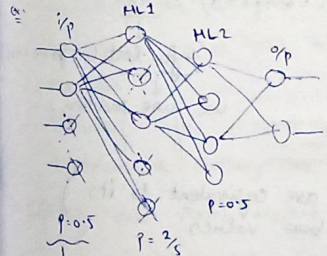
(weights initialised from a normal distribution where mean = 0 & std. dev = σ)

3) Kaiming He Initialization:

• He Uniform: $W_{ij} \approx$ uniform distribution $\left[-\sqrt{\frac{6}{input}}, \sqrt{\frac{6}{input}}\right]$

• He Normal: $W_{ij} \approx N(0, \sigma)$; $\sigma = \sqrt{\frac{2}{input}}$

Dropout Layer:

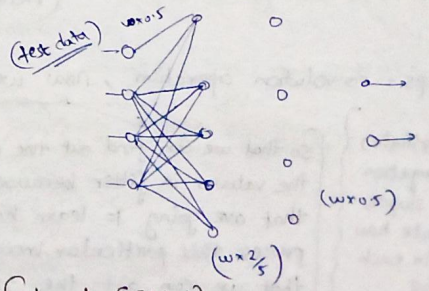


(means at a time, only 50% of neurons will be active)

(only '2/5' % of neurons are active)

• We deactivate some neurons along with this, we are providing some feature samples so that the entire model doesn't overfit (wrt training)

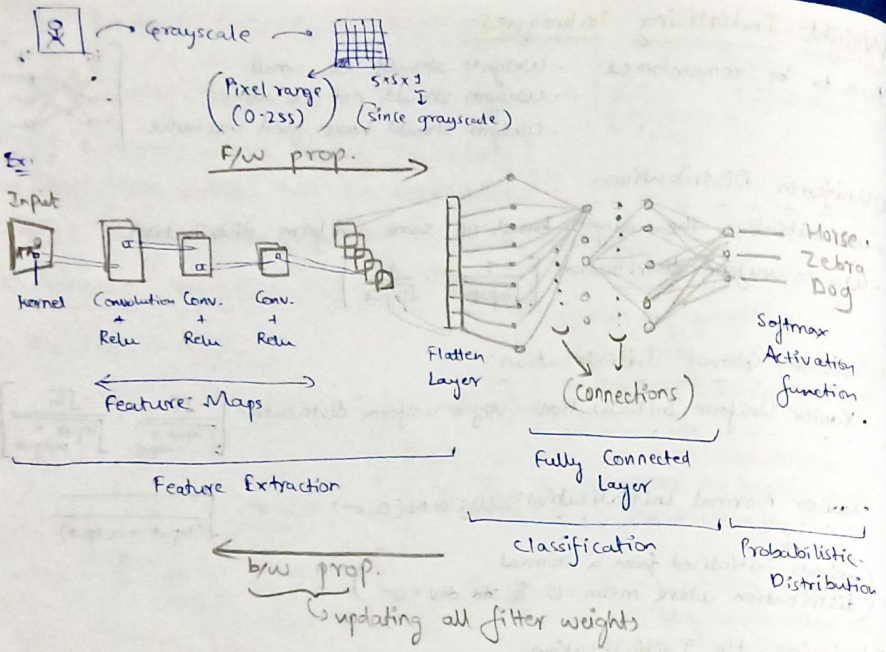
• Wrt test data, all the neurons will get interconnected & weights are multiplied with probability.



Convolutional Neural Network (CNN):

1) ANN → supervised learning → Classification / Regression

∴ 2) CNN: i/p = images (eg: image classification, object detection.)



Convolution Operation:

1) We normalize the pixels (divide by 255)

2) We have a 'filter' matrix → which is specialised in a task.
(ex: identifying vertical / horizontal edge)

3) Apply filter on the image → final matrix
(let 3×3) (let 6×6) (n−f+1 = 6−3+1 = 4) ∴ 4×4
Info. lost (as prev. = 6×6)
↓
we use padding.

4) Padding → Add another layer
(padding)
→ Zero padding = (values = 0)
→ Neighbour padding = (values are equivalent to its neighbour values)

5) Above steps = convolution operation, now we apply 'ReLU'.

So that we can find out the derivative & we can update the values in filter because these filters are the one that are going to learn how we will be able to process this particular image in an efficient way so that we can get the right kind of op.
(We can have many)

∴ Convolution Layer = Conv. operation + 'ReLU'

*We need derivates in backpropagation to calc. the slopes, which indicate how much wt. in each NN should be adjusted to minimize the loss func. & improve the model's performance.

Pooling: focuses on summarizing & retaining the most imp. features from the feature maps.

3 types of pooling:
- max pooling → selects max. value in each patch of the feature map
- min pooling → selects min.
- mean pooling → selects avg.

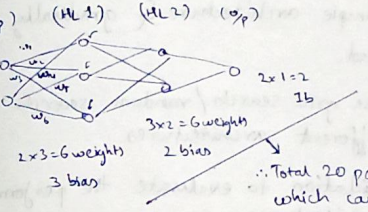
(Refer Pdf for more)

END TO END DEEP LEARNING PROJECT USING ANN

1) 'pip install -r requirements.txt'

2) Create 'experiments.ipynb' ⟹ Pre-process the data (use LabelEncoder, OHE & train_test_split)

3) ANN → Sequential Network



4) • When we start creating our ANN, we have to initialize a sequential network.
- Dense - used to create hidden neuron
- We apply activation function to every node (ex: sigmoid, ReLU, etc)
- Optimizer - useful in back propagation because these are responsible in updating the weights
- Loss func. → we have to reduce this
- Metrics
- Training info. - store the 'logs' in a folder, so that I can use Tensorboard (displays the logs)

5) Building our ANN model
→ model = Sequential([(Connected with i/p layer, so we need to specify the i/p shape)
  Dense(64, activation='relu', input_shape=(X_train.shape[1],)), (HL1)
  Dense(32, activation='relu'), (HL2)
  Dense(1, activation='sigmoid') → o/p layer
])
model.summary()

6) Compiling & training our model.

7) Setting up Early Stopping & Tensorboard

8) In 'prediction.ipynb':

→ Load the ANN trained model, scaler, pickle, onehot

→ Take a sample i/p → convert it into suitable format i.e OHE few colums & scale it.

→ Predict on the i/p.

9) Write 'app.py' → deploy on streamlit

10) Practice by predicting a value now i.e. practice regression prob. stmt. on this same dataset. (Similar to prev.)

11) Determining the optimal number of hidden layers & neurons for an ANN:

↳ Guidelines:

→ Start Simple: Begin with a simple architecture & gradually increase complexity if needed.

→ Grid Search/Random Search: Use grid search / random search to try different architectures.

→ Cross-Validation: Use cross-validation to evaluate the performance of different architectures.

→ Heuristics & Rules of thumb: → A common practice is to start with 1-2 days.

↓

The no. of neurons in the hidden layer should be between the size of the i/p layer & the size of o/p layer

→ Refer 'hyperparameter tuning.ipynb'.

// Refer files for above project.

NLP IN DEEP LEARNING

Simple RNN → LSTM/GRU RNN → Bidirectional RNN → Encoder Decoder

↓

Transformers ← Self Attention