Large Language Model

~~copy list with random Pointer~~

king - Man + Women
king Minus Man Plus women is Queen

Statements like this needs semantic meaning to be preserved
This can be done

I) Tokenize the given input into subwords such as king, women, queen, plus, minus
II) Through positional encoding order of the words are preserved
III) The semantic meaning can be inferred from a cluster of similar sentences.
IV) This semantic meaning must be retrieved through methods like retrieval augmented generation (RAG)

Chapter I: Understanding the LLMs

What is lang AI?

What are AI system
AI is the process of developing intelligent computer programs so that it can understand (human) AI and replicate it. The AI systems are pre-trained with a large amount of data

Language AI is the subfield of AI that focuses on developing technologies capable of understanding processing and generating human language

Language AI models are pre trained on large corpus of data therefore they are called as large language models

Heaps and Priority Queue

The non-transformers models are simple LM which can process and structure the input using simple vector representation or neural networks
eg - Bag-of-words, word2vec, attention etc

They donot perform well if i/p sentence is large
Transformer based models rely on transformer ~~framework~~ in order to process the input the attention mechanism in transformers makes it suitable to process longer input sequence
ex - BERT, GPT, T5, switch

Representing language as a bag of words

A sentence comprises of several words. each word is again comprised of characters each word is seperated from another by a whitespace.

vocabulary
i/p → tokenization → Vector Embedding

Tokenization A large i/p is broken into smaller units called tokens. Every time we get an i/p it must be broken into tokens and these are added into data dictionary called the vocabulary. This vocabulary is the reference for any language models

only unique tokens are added to the vocabulary (BOG)

"Bag of words" → 3 tokens (word-level tokenization)

i/p: That is a cute dog i/p: My cat is cute

that	is	a	cute	dog	my	cat
1	2	1	2	1	1	1

7 vocabulary

NOTE:

All though repeated words are not added to vocabulary but the vocabulary keeps tracks of how many time a word is used in input

Vector Representation of Bag-of-words

Each token must be converted to a numerical representation this numerical representation allow the mapping of the i/p to the vocabulary

The Embedding technique used in 'bag of words is called one-Hot Encoding where there is a 'zero' in the vocabulary if the word is not present in the input

eg- i/p my cat is cute

that	is	a	cute	dog	my	cat
↓						
0	1	0	1	0	1	0

one-Hot encoding is a fixed length encoding technique

0	1	0
0	0	0
1	0	1

0's no meaningful content

1's content

sparse matrix

One-Hot Encoding contains a large no of 0's which is computationally expensive. The biggest drawback is that it doesn't contain any semantic

Bow cannot prenure the semantic meaning of word

Advantages and Disadvantages of BOG

Advantages - I) Easy to use
II) Used as base for newer models

Disadvantages I) If there are no whitespace the BOW can't generate

Dense vectors

Instead of storing embedding as sparse matrix we try to store the embeddings as a dense vector
The most popular dense vector technique are-
I) word2vec II) GloVe

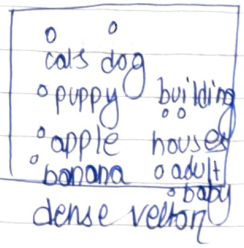
Dense vector Representation

Dense vectors are embeddings of tokens which try to preserve the semantic meanings of words
Dense vectors are created using Neural Network
ANN is adjust with some features and some weights

In Dense vector representation there are a large no of dimes Each word in the input has a similarity score for every dim

	cats	puppy	houses	apple	baby	
animal	91	.93	-56	-67	01	Number of dimensions
newborn	-31	.71	-32	-1	90	
human	-91	.36	31	29	87	
plural	94	94	94	51	-31	
fruit	-51	-5	-5	89	-31	

Embedding of words that are similar will be close to each other in dimensional space



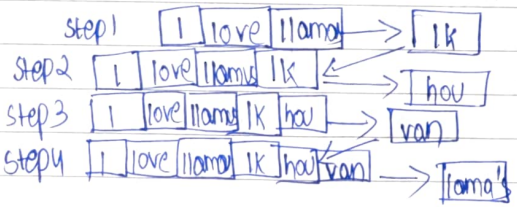
dense vectors

Attention (RNN with Attention)

Although dense vectors can preserve the semantic meaning of the input sequence. It doesn't perform efficient when the context window is large.
To overcome this advantage RNN (Recurring Neural Network where proposed

RNN are Neural Networks which can model sequences as an input, this helps preserve the contextual meaning as well as the input sequence

RNN- to capture semantic meaning

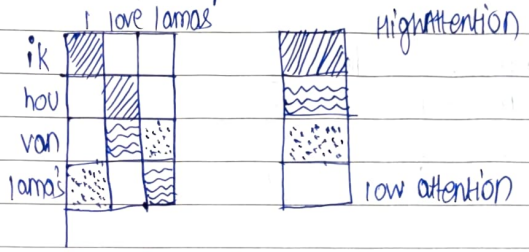


To remember a RNN in the i^{th} step pases the output of previous 1 to i-1 steps along with the input.

The disadvantage of a recurrent neural networks is that a large no of output words has to be passed everytime which is computationally very expensive
To reduce this computational complexity the attention mechanism was produced.

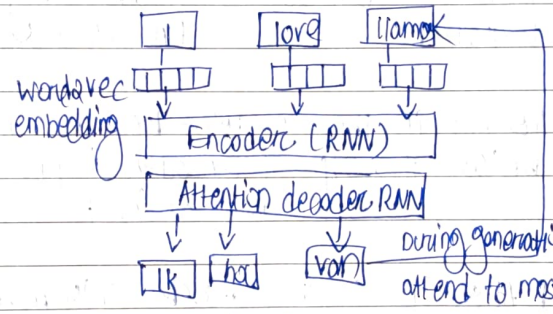
Attention mechanism

Attention allows the model to focus on the specific parts of the input that are relevant to each other.



From the attention matrix the model can predict which is the most relevant part of the output
Therefore it can preserve both meaning and sequence.

RNN + Attention Mechanism

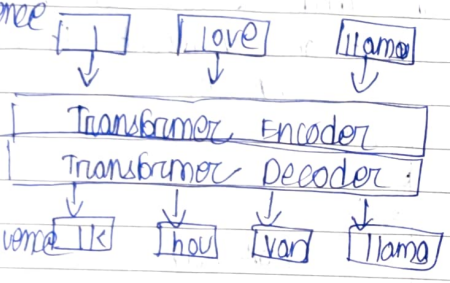


Using RNN + Attention we can maintain sequence for large context this helps us in translation related work.
Therefore these models are known as representation mode

Transformers

A Transformer is a neural network that consist of 2 layers
(i) Encoder layer
(ii) Decoder layer

A transformer model can be trained in parallel unlike RNN model.



Encoders & Decoders

Encoding

The encoder takes a sequence of tokens and transforms it into a series of contextualized embeddings.

These embeddings capture the semantics meanings as well as the sequences

There are 2 layer - Self Attention
- Feed Forward

Decoding

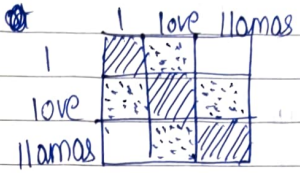
Decoders can be used in 2 scenarios - Decoder only
- Encoder & Decoder

In the encoder-decoder model the o/p of encoder is provided to decoder as input

In the decoder only model there is no encoder the i/p is directly given to the decoder

Self Attention

Self Attention is at the heart of transformed model the self attention is different from the previous attention

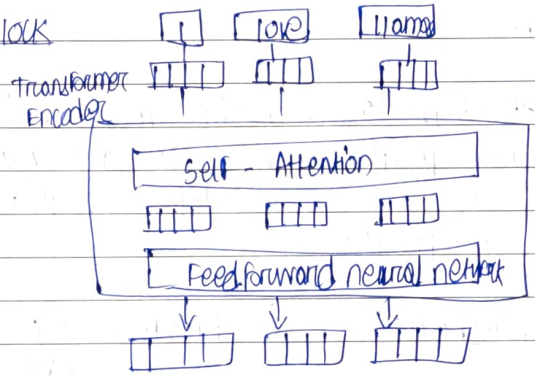


In SA the attention b/w every token in the i/p is calculated. No o/p is involved in case of self Attention

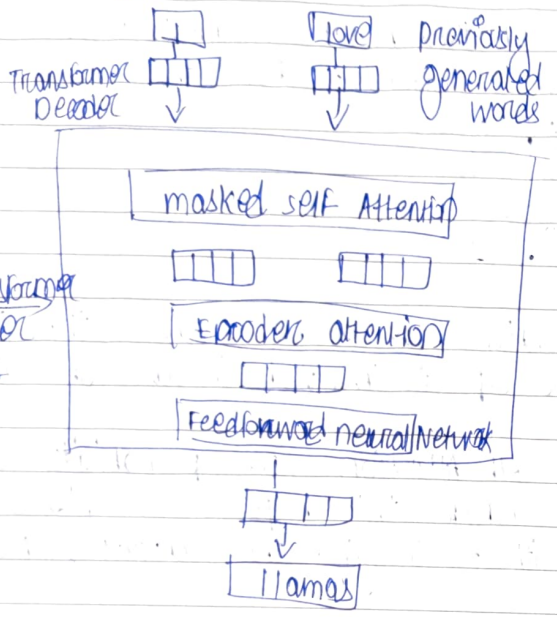
Because of the self Attention low probability token in inputs are discarded, this helps us to provide input to the model without changing the contextual meaning of the input.

This SA mechanism led to the creation of Generative Model.

Encoder Block



Decoder Layer



The decoder has an extra layer for encoder attention to pay self-attention to the encoder output masked.

The Self-Attention layer in the decoder is masked so that future positions are not available during decoding otherwise the model will just copy the input instead of predicting

Types of Transformers
(I) Encoder only
(II) Decoder only
(III) Encoder-Decoder

Bidirectional Encoder Representations from Transformers

Generative or completion LLM

User query (prompt): Tell me something about llamas

Generative LLM
Task: complete the i/p

output (completion)

Decoder only model

Context window: represents the max no of tokens that a model can process.
A large context window allows entire document to be passed through the LLM.

Pretrained LLM

Before a LLM is released in the public it is trained on massive purpose of text data these are called pretrained LLM.

representative model

Bag of words (No sequence or context)
↓
word2vec (Preserves semantic meaning but doesn't preserve sequence)
↓
RNN (can remember seqence the partial output passed as i/p
↓
RNN+Attention → Transformers capable of representation as well as generation
No need to pass o/p like RNN it can remember sequences through attention
contain encoder / decoder or both

Fine tuned LLM

A pretrained LLM that is available in the public can be fine tuned on certain parameters to perform specific tasks

Tokens and Embeddings

Input: Have the bards who preceded

↓ Tokenisation
Breakdown the text into smaller pieces (words or parts of words)

Tokens: [Have] [the] [bards] [who] [preceded]

↓

Embedding
Turn tokens into numeric representations capturing their meaning

The LLM although they process and generate text can't understand characters in the text
Therefore characters in the input to LLM must be converted to numeric value.

Converting the entire i/p sequence to numeric values doesn't make the conversion efficient

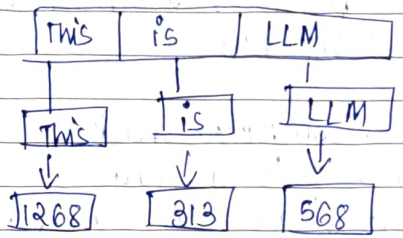
A better way to convert text into numerical values is to break the i/p into smaller units and create numerical representation of those units.

NOTE - All the open source LLMs are available in a python class called transformers

The input id variable stores the token ids of all the generated tokens.

Input and outputs of LLMs

When the i/p sequence given to the LLM it doesn't process the character tokens rather it process numerical representations of those tokens.



In addition to being used to process the i/p text into a language model, tokenizers are used on the output of the language model to turn the resulting tokens into the o/p word for token associated with it.

Input: Have the bards who preceded

Tokenizer: encode

Tokens: [Have] [the] [bards] [who] [preceded]
Token ID: [6475] [278] [27518] [3163] [1058]

↓

Language model

↓

output token ID: 1394

Tokenizer: decode ↓

[on]

When a model produces o/p it doesn't produce a sequence of text rather it produces a sequence of numerical values. These numerical values

must be converted to characters. This is done by mapping the token ID to its corresponding text.

Properties of Tokens

i) The first token is ID 1 (<s>) a special token indicating the beginning of the text

ii) some tokens are complete words
iii) some tokens are parts of words
iv) Punctuation characters are their own token
v) space characters does not have its token

what are Tokens?

The smallest units of text that a model processes is known as a token input sequence is broken into these small units.

Types of Tokenization - word
subword
character
Byte

Tokenization methods

Word level tokenisation - The process of splitting i/p sequence on text into individual words

Advantages - simplifies the input by dividing them into words. Preserves the contextual meaning of each token. works vor with languages where clear word boundaries is defined such as English

Disadvantages - similar words with slightly different meaning are stored in the vocabulary as diff tokens
This increases size of vocabulary
ex - apology, apologies, apologetic

ii) word level tokenization has no mechanism to detect OOV

OOV (out of vocabulary) :- when the LLM model is pretrained the vocabulary is created. when the tokenizer finds a token which is not in the vocabulary it is called OOV token

character-level tokenization - when the i/p sequence is broken in individual characters including letter, numbers, digits, spaces

advantages - → No OOV words issues
it works well with creative and noisy texts
it is more fine-grained (useful for spelling correction)

Disadvantages - it looses context when the i/p sequence is long
loss of semantic and contextual meaning because characters can't hold semantic meaning like words

Byte-level tokenization - breaks the i/p sequence to individual bytes that are used to represent unicode characters

This allows for universal representation including texts, numbers and emojis.

advantages
- No unknown tokens

DOMS

Size of vocabulary is very less - 1 byte = 256 token used bc social media, coding, multilingual input

Disadvantages - The semantic and contextual meaning gets lost because of unicode representation

word-level	preserves semantic meaning	large vocab OOV
character level	more fine grained	it doesn't preserve semantic
Byte-level	universal representation of language	

Subword - level tokenization can produce words, character tokens, it is middle ground of character and word

it is useful for designing generative models like GPT

advantages - it handles OOV words reduces the vocab size

Improves generalization across word forms

Disadvantages - it may produce meaningless subwords

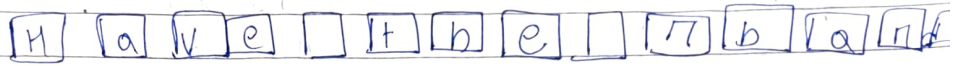
Requires pretraining to built this vocabulary

complex to implement than word-level or character level

word tokens

subword tokens

character tokens



Byte tokens

H	a	v	e	<space>	t	h	e	<space>	Π
0	0	0	0	0	0	0	0	0	1 1 1 1
1	1	1	1	0	1	1	1	0	1 0 0 0
0	1	1	1	0	1	1	1	0	1 0 0 1
0	0	1	0	0	1	0	0	0	1 1 0 1
1	0	0	0	0	0	1	0	0	0 1 1 0
0	0	1	1	0	1	0	1	0	0 1 1 1
0	0	1	0	0	0	0	0	0	0 1 0 0
0	1	0	0	0	0	0	1	0	0 1 0 1

Problems solved by Tokenization	How tokenization helps
words vary in form (run, running)	Breaks into subwords captures shared meaning
Rare or new words (photosynthesis)	splits into known chunks → avoids unknown tokens
Misspelling as typos	subwords unit still makes sense (transformers)
Multilingual text	Uses shared vocabulary across languages
Model i/p size limits	controls token count – fits within model window

DOMS

Tokenization

- Character level
- word
- subword
  - BPE
  - word piece
  - sentence piece
- Byte

Byte pair encoding is a subword level tokenization algorithm
It begins with character level merges to form subwords

Requires - corpus of words, No of merges
output - Tokenized corpus, subword vocabulary

Preprocessing
For each word in the corpus to split the word into characters
Append EOW symbol end For.

Tokenization
Initialize the vocabulary with unique characters & EOW
For merge step = K to N
do
count Frequency of all adjacent symbol pairs
Identify the most Frequent pair (a,b)
Merge (a,b) → ab and add to Frequency
End For.

① aa_ abc_ abc

② 'a','a', 'a','b', 'c', 'a', 'b', 'c'

③ (a,b) – 2
(b,c) – 2
(a,b) → ab

vocabulary
'a', 'b', 'c', '_'

DOMS

Initial corpus: ["aa", "abc", "abc"]

"aa" → ["a", "a", "_"]
"abc" → ["a", "b", "c", "_"]
"abc" → ["a", "b", "c", _]

Step2: Iterative Merging

Most frequent pair: ("a", "b")
merge into: "ab"
corpus update: ["a", "a", "_"]
["ab", "c", "_"]
["ab", "b", "c", "_", "ab"]

Most frequent pair: ("ab", "c")
merge into: "abc"
corpus update: ["a", "a", "_"]
["abc", "_"]
- ["abc", "_"]

vocabulary: ["aa", "b", "c", "_", "ab", "abc"]

Iteration 3:
Most frequent pair: ("a", "a")
merge into: "a"
corpus update: ["aa", "_"]
["abc", "_"]
["abc", "_"]

vocabulary: ["a", "b", "c", "_", "ab", "abc", "aa"]

Final Token: ["aa", "_"]
["abc", "_"]
["abc", "_"]

Final vocabulary: ["a", "b", "c", "_", "ab", "abc", "aa"]

Byte pair encoding splits the input into words and splits those words into characters.
The unique characters are added to the vocabulary
From the input corpus it finds out the pair of most occuring characters (a - byte pair)
and merges them.
This merged pair of characters is called as a ~~character~~ subwords and in future steps may be combined with other subwords.

For ex- a,b merged to ab
ab,c ↓ abc

These subwords are added to vocabulary in each step and the final vocabulary as well as final tokenized corpus is returned.

Using Byte pair encoding find the final vocabulary and the final tokenized corpus

h,ug P,ug P,un B,un h,ug,s
{hug,10}, {Pug,5} {Pun,12}, {Bun,4}, {hugs,5}

vocab
h, u, g, P, n, B, s

step1: u,g - 20 → ug.
u,n - 18 16 → (un is now token)
Pu - 17 12 (ug is now a token)

h,ug P,ug P,un B,un h,ug,s
h,ug → hug

Wordpiece - is a subword level tokenization method developed by google used by BRET

Like byte-pair encoding word piece also merges tokens to form the training vocabulary

Unlike byte-pair encoding which merges the most frequent pair of characters word piece follow a merging rule

$$\text{Score} = \frac{\text{frequency\_of\_pair}}{\text{frequency of first ele} \times \text{frequency of second ele}}$$

{un} {able}
{nu} {aging}

By dividing the frequency of the pair by the product of the frequencies of each of its parts, the algorithm prioritizes the merging the pairs where individual parts are less frequent in vocabulary.

it wont necessarily merge ("un", "able") even if the pairs occurs very frequently in the voulary because the two pairs un and able will likely each appear in a lot of other words and have a high freq

Word piece prioritizes merging the subwords that are less likely to appear as individual tokens throughout the vocabulary

once the vocabulary is created using the merge rule wordpiece follows the given greedy strategy

DOMS

1. Start at the beginning of the word
2. Find the longest substring that matches a token in the vocabulary
3. Add that token to the output
4. Move forward in the word and repeat until the entire word is tokenized

NOTE - This algorithm is for a single word and must be repeated until all words in the input sequence are tokenized

Wordpiece appends the symbol ## before any subword to specify that the particular subword is part of another token.

Ex - vocabulary = {"un", "##aff", "##able", "##ably"}
word = "unaffable"
match = "un"
Remaining = "affable"

M = "## aff"
R = "able"
M = "##able".

Final = {"un", "## aff", "##able"}

unknown

NOTE - wordpiece uses the UNK symbol to represent input subwords that do not match any token in the vocabulary.

vocabulary = {UNK, a, aa, ##b, ##c}

aa → aa
abc → ##bc → ##c
match match match

DOMS

vocabulary = {UNK, a, aa, ab, abc, ##b, ##c}

corpus = {"aa", "abc", "abc"}
match match match

Sentence piece

is a subword level tokenization algorithm that is language independent it can work well with languages which do not have clear boundaries such as Mandarin (chinese)

unlike byte pair encoding / wordpiece that create the vocabulary by merging tokens sentence piece requires a large vocabulary to start.

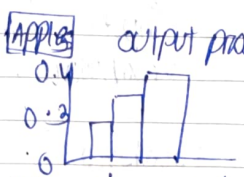
It introduces a special metacharacter to represent the spaces, the most used ex- is unigram model

UNIGRAM MODEL

The unigram model is used in LLM models like T5, BigBird, XL Net.

INPUT TEXT ⟶ BAG OF WORDS

I like apples
INPUT TEXT



Since it produces language independent or universal tokens ... so it is called unigram

The unigram model uses viterbi algorithm to find the most probable segmentation of a token i.e the sequence of tokens that maximizes the product of their unigram probability

This probability is not the raw frequency rather it is a learned probability that the model learns based on how frequently the unigram occurs on the vocabulary.

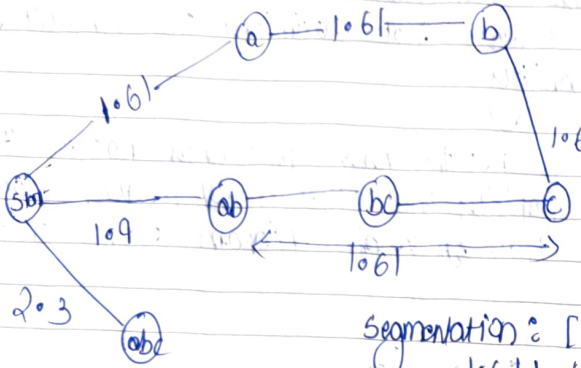
viterbi decoding in a unigram

Cost of a segment is the negative logarithmic of the probability of its constituent tokens.

$P_{abc} = P_a + P_b + P_c$

The viterbi algorithm finds the most probable by finding segmentation which incurs the least cost.

token	probability	cost - log p
a	0.20	1.61
b	0.20	1.61
c	0.20	1.61
ab	0.15	1.90
bc	0.15	1.90
abc	0.10	2.30



Segmentation : ["a", "b", "c"]
1.061 + 1.061 + 1.061 = 4.083

Segmentation : ["abc", "c"]
cost 1.90 + 1.061 = 3.51

Segmentation : ["a", "bc"]
cost : 1.061 + 1.90 = 3.51

Segmentation : ["abc"]
cost: 2.30

and
Given the vocabulary the probability according to viterbi the most probable segmentation is abc

The probability of subwords present in the vocabulary are not fixed, they change over time depending on the input corpus therefore these probabilities are called as learned probabilities. in the unigram model

The probability of a segment (abc)

P_{abc} is the sum of probabilities of its constituent segments

$$P_{abc} = P_a + P_b + P_c \text{ or } P_{ab} + P_c \text{ or } P_a + P_{bc}$$

The cost of a segment C_{abc} is the negative log of probability $C_{abc} = -\log P_{abc}$

These probabilities change over time based on the likelihood of a subword appearing on the training corpus

Ex corpus - aa abc abc

Token	probability
a	0.1
aa	0.3
abc	0.6

1. [aa] [abc] [abc]
2. [a, a] [abc] [abc]

These are only valid segmentation using the given vocabulary.

Segmentation 1 : $0.3 \times 0.6 \times 0.6 = 0.108$
Segmentation 2 : $0.1 \times 0.1 \times 0.6 \times 0.6 = 0.0036$

Normalised probability - $\frac{0.108}{0.1116}$

$\frac{0.0036}{0.1116}$

once the normalised probability is calculated unigram uses the EM (~~Estimation~~-Maximization) algorithm to update the probability. Expectation

Finding the Expected count for each subword (E-word)

subword	count in seg-1	Count in segmentation-2	Expected cant
a	0	2	0x0.968 + 2x0.032
aa	1	0	1x0.968 + 0x0.032 = 0.968
abc	2	2	2x 0.968 + 2 x 0.032 = 2.000

Maximization step

The maximization step the probability the subwords are updated on if they appear on the input corpus

0.064+ 0.968 + 2.000 = 3.032

subword	expected count	updated probability	calculation
a	0.064	0.0211	0.064 ÷ 3.032
aa	0.968	0.3193)	0.968 ÷ 3.032
abc	2.000	0.6596	2.000 ÷ 3.032

As seen the probability of tokens 'aa' and 'abc' increased and 'a' decreased because the

a did not occur
aa occured once increased slightly
abc occured twice so it increased

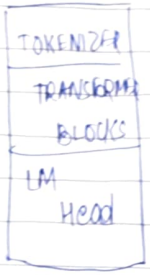
Final Tokenization using Unigram

Given the initial learned probabilities and the subwords unigram model finds the most probable segmentation using the ~~word~~ viterbi algorithm then using EM algo

so for the given vocabulary the best tokenization is "aa abc"

[Prev My Copy]

Input - Write an email to Sarah



Vocabulary

Dear	50%
a	20%
not	30%
this	20%

The i/p passes into the tokenizer there is broken individual tokens. These tokens then passed through transfer block which changes the probability of tokens in the vocabulary once the probability distribution of the vocabulary changes LM had selects the next token based on the decoding strategy

LM had selected one token from the vocabulary this is called as decoding. The easiest decoding strategy is to select that token which has highest probability this is known as greedy decoding.

Softmax function transforms array of values to its corresponding probability distribution such that all the o/p values remain between 0 to 1

$$\text{Softmax} = \frac{e^{z_i}}{\sum_{j=1} e^{z_j}}$$

$z = [2.0, 1.0, 0.1]$

$e^{2.0} = 7.39, \quad e^{1.0} = 2.72, \quad e^{0.1} = 1.11$

$\sum e^{z_j} = 7.39 + 2.72 + 1.11 = 11.22$

DOMS

$\text{Softmax}(2.0) = \frac{7.39}{11.22} = 0.659$ $\quad \text{Softmax}(1.0) = \frac{2.72}{11.22} = 0.242$

$\text{Softmax}(0.1) = \frac{1.11}{11.22} = 0.099$

Let us calculate attention weights for the word crossed in the i/p sentence the chicken crossed the road

Let us assume dimension $d_A = 4$ therefore the scaling factor $\sqrt{d_K} = 2$

$Q_{crossed} = [1, 2, 1, 0]$

key vectors for the token -

$K_{The} = [1, 1, 0, 0]$
$K_{chicken} = [2, 1, 1, 0]$
$K_{crossed} = [1, 2, 1, 0]$
$K_{The} = [1, 1, 0, 0]$
$K_{Road} = [1, 1, 1, 0]$

dot product $Q_{crossed}$ by each k_i

$Q \cdot K_{The} = 1x1 + 2x1 + 1x0 + 0x0 = 3$
$Q \cdot K_{chicken} = 1x2 + 2x1 + 1x1 + 0x0 = 5$
$Q \cdot K_{crossed} = 1x1 + 2x2 + 1x1 + 0x0 = 6$
$Q \cdot K_{Road} = 1x1 + 2x1 + 1x1 + 0x0 = 4$

Scaled The $= \frac{3}{2} = 1.5$
Scaled chicken $= \frac{5}{2} = 2.5$
scaled crossed $= \frac{6}{2} = 3.0$
scaled Road $= \frac{4}{2} = 2$

DOMS

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

$z = [1.5, 2.5, 3, 2]$

$e^z = [4.48, 12.18, 20.08, 7.38]$

$\sum e^z = 44.12$

Softmax $(z) = [0.101, 0.276, 0.455, 0.167]$

Crossed & The $= \frac{4.48}{44.14} = 0.10$ ✗

Crossed & Chicken $= \frac{12.18}{44.14} = 0.28$

Crossed & Crossed $= \frac{20.09}{44.14} = 0.46$ ✓

Crossed & Road $= \frac{7.39}{44.14} = 0.17$

Crossed gives highest attention itself followed by chicken & road gives least attention the token The.

We are assuming the value of $v=1$

Positional Embedding

Using the FFNN and the attention mechanism the model can understand contextual relationship bet^n the i/p token but it can't remember sequence of i/p token therefore it uses positional embedding to differentiate and remember the sequence of each i/p token.

most used embedding is ~~RoPE~~ RoPE. (Rotary Positional Embedding)

RoPE doesn't add positional embedding vectors.

RoPE rotates the query and key vectors for each token

" " " " " " " using Trigonometric functions. each tokens vectors is rotated by an angle based on its position.

The dot product between rotated vectors encode relative position

Decoding in RoPE

Each token's query & token vector is rotated based on its position

During decoding, the model compares these rotated vectors to compute attention scores.

The angle betn vectors how far apart tokens are.

Relative Position Decoding

$n - m = 2$ tokens part

Text Classification using LLMs (Ch-4)

Text classification

Input - This is a good movie

Pre determined
- positive 80%
- Negative 10%
- Neutral 10%

Given an input statement or text, the process of classifying it into predetermined class is called as text classification

It is useful in scenarios where there is a large amount of unorganized text that must be classified & organised one of the examples is called sentiment analysis

Text classification is useful in SA, review classification document classification)

Traditionally Text classification is done using ML and DL model but since the introduction of transformer NN, JC is been done using LLM

Text classification - ML — word2vec, Glove
DL — LTM, GRU

Input → LLM - +ve
- -ve
- neutral

Metrics - A classification model performance can be evaluated by using certain evaluation metrices

There are 4 basic metrices/ primary metrices from which evaluation metrices are defined - TP, TN, FP, FN

	Actual values: +ve	Actual values: -ve
Predicted values: +ve	TP	FP — -ve review incorrectly classified as +ve
Predicted values: -ve	FN — +ve review incorrectly classified as -ve	TN — -ve review correctly classified as -ve

+ve review correctly classified as +ve

Confusion Matrix - From confusion matrix we can derive the evaluation metrices such as accuracy precision, recall, F1 score

Accuracy → $\frac{TP+TN}{TP+TN+FP+FN}$

Precision → $\frac{TP}{TP+FP}$

Recall → $\frac{TP}{TP+FN}$

F1 score - $\frac{2 \times Precision \times Recall}{Precision + Recall}$

Support - No of samples

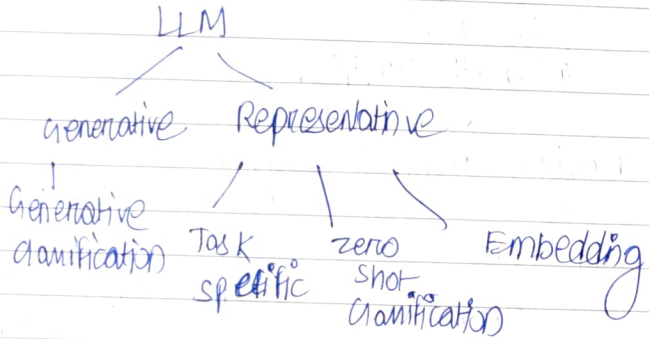
(Macro, weight) Avg

Precision - measures how many of the items found are relevant, which indicates the accuracy of the relevant results

Recall - refers to how many relevant classes are found which indicates its ability to find all relevant results

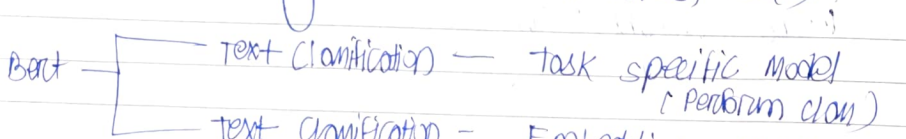
Accuracy - refers to how many correct predictions the model makes out of all predictions, which indicates overall correctness of the model

F1 score - balances both precision & recall to create a model's overall performance (categorised as best metric)



TC using Representative LLM

Classification with predefined representation models generally comes in 2 flavours either using a task specific model or an embedding model (when training dataset are labelled, used)



During Training if labelled datasets are not provided then representative models can do a 3rd type of classification

Task specific Model

It is a fine tuned variant of the base representative model such as "bert-base-uncased-finetuned-SST-2" which is best fine tuned on the labelled dataset standford sentiment Treebank

Steps in TSM →

1) The input text is tokenized into subword units (Model i/p)

2) Model forward pass - The forward pass process the tokens finds semantic meaning

3) Classified Head - Based on semantic meaning of token the model has classification layer that outputs probabilities based on semantic meaning

4) Prediction - The class with highest probability is chosen

Embedding Model - In case we don't task specific model ie freely available or if we find user defined control on classifier in that case we can use a embedding model for text classification

Steps →

1) Load an Embedding model - for example sentence transformers / all - mpnet - base - v2

2) Convert text to ~~strings~~ Embeddings - The model generates a vector for each piece of text

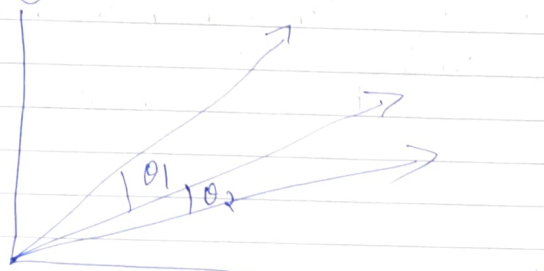
3) Train a classifier - Feed these vectors into a standard classifier

4) Predict - for new input, create the embedding and then run it through the classifier

Zero shot classification

When there are no labelled datasets, then text classification can be done using representative LLM's through zero shot classification

Step 1 - Create a text discipline of each label
Step 2 - Embed the label description and your documents using the same embedding model
3 - Compute the cosine similarity b/w each document embedding and each label's embedding
4)



In cosine similarity, the similarity between 2 embedding is found using trigonometric function cosine the angle cos θ determines which 2 embedding are closer

Ex - A very +ve movie review
A very -ve movie review

Text to classify - This movie was an absolute masterpiece

compute similarity with each label, ~~whichever~~ which is higher gets assigned

Text classification using generative LLMs

Flan - T5 for classification -

T5 -> Text through text Transfer Transformer
Flan - T5 is a fine tuned variant of T5 esp for text classification

1) Load the model in your python program
2) Create a prompt - Is following sentence +ve or -ve
3) Generate output - the model's textual o/p might be the word +ve / a -ve
4) Convert o/p to a label - Map - +ve -> 1
-ve -> 0

Text classification using a closed source model (chatGPT)

A closed source model provides API for task specific jobs like JC

Ex -

Document - the acting was superb and I laughed the whole time

Expected o/p : "1" (box its +ve)

Ch-5

Text Clustering & Topic Modeling

Text clustering - Given a large no of text inputs, the process of text clustering is to group similar documents into a cluster based on their content and semantic meaning. These groups are called as clusters and contain similar documents.

Clustering helps us in organising a large amount of unorganised input.

Applications

information retrieval - when the text is organised into clusters it is easier to retrieve data items and other related data items

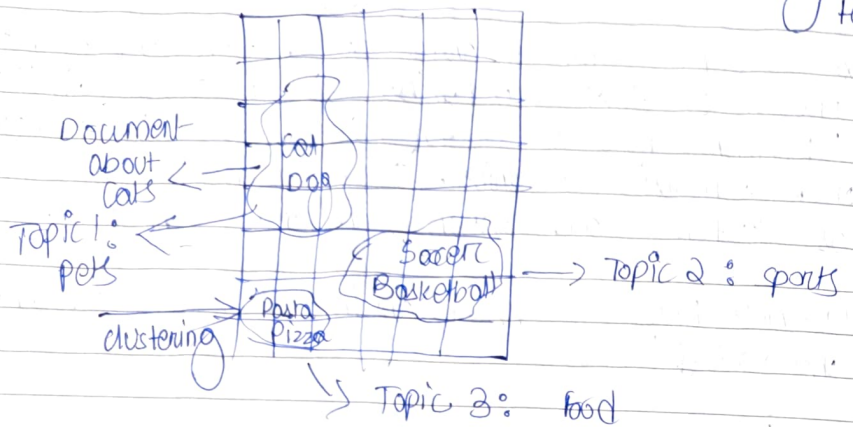
Recommender system - when the user selects a particular data item all other data items in the cluster can be provided as recommendation

spam filtering - when a menage is labelled as spam all other similar menages are also added to the spam clusters

Text summarization - Group similar sentences in a long text can help create a summary of the text

Input (Textual data)

output (cluster of semantically similar text)



Topic Modelling

Every cluster is created must be given an abstract definition which describes all the data items in that cluster. This abstract definition of a cluster is known as topic and process of doing so is called Topic Modelling

A Topic helps in identifying a cluster as each topic is associated with several keywords

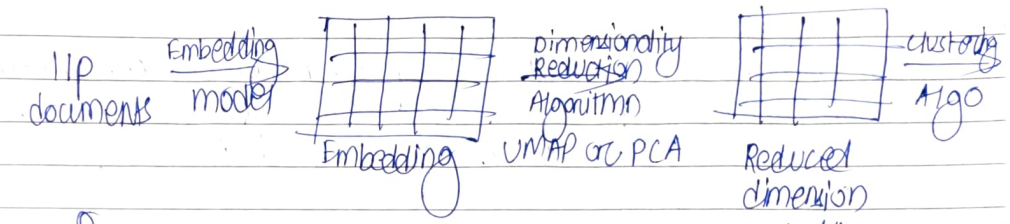
In this chapter we study the basic pipeline for text clustering and also discuss a fine-tuned model specifically made for topic modelling called as Bert topic

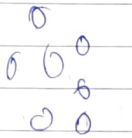
A common pipeline for text clustering

Traditionally Text clustering is performed using ML & DL methods but since introduction of transformer architecture, text clustering is being done using LLMs

There are 3 steps which are executed for text clustering using LLM.

1. Convert the i/p documents into embedding with an embedding model
2. Reduce the dimensionality of embeddings with a dimensionality reduction model
3. Find groups of semantically similar documents with a cluster model





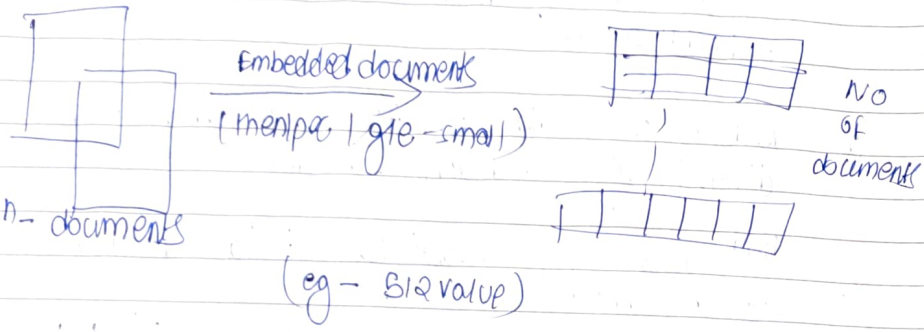
Clustering

Step 1) Embedding the input documents

Textual info / data is converted to its corresponding numerical representations known as embeddings these embeddings can remember the contextual meaning of words

Embeddings for TC can be created using a general purpose embedding model called as "thenlper / gte-small"

Embeddings that are produced by ↑ have a very high dimension



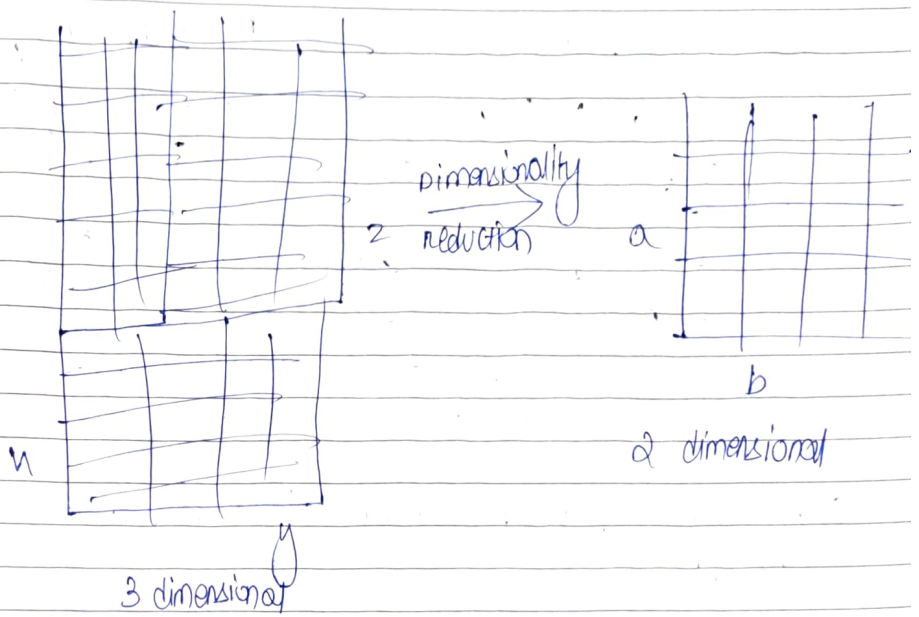
Step2) Reducing dimensionality of embedding - An embedding of dimension D takes ed time to retrieving the embedding vector when dimension d increases the surface increases

It is computationally expensive, therefore the dimensions of embeddings created in step 1 must be reduced

Dimensionality reduction is the process of reducing the dimensions and still getting o/p as if no dimension is reduced

2 of the dimensionality reduction is called->

3 dimensional space
2 dimensional space

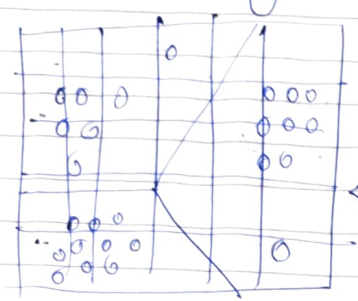


step3) Cluster the Reduce Embeddings - A clustering algo takes the reduced or compressed dimension embedding & produces clusters out of them

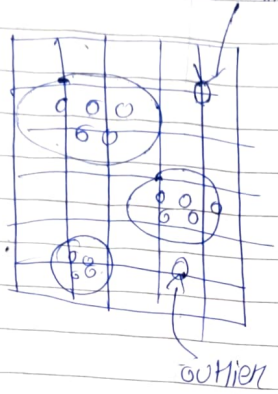
Centroid Based	Density Based
In this approach one data item is choosen as centroid all other data item that surrounding the centroid within a threshold distance is included in cluster	In this approach words have similar embeddings are put into a cluster
These approaches can't detect outliers	These approach can detect outliers

NOTE - outliers are those data items which don't belong to any cluster

Ex - K mean clustering



centroid based



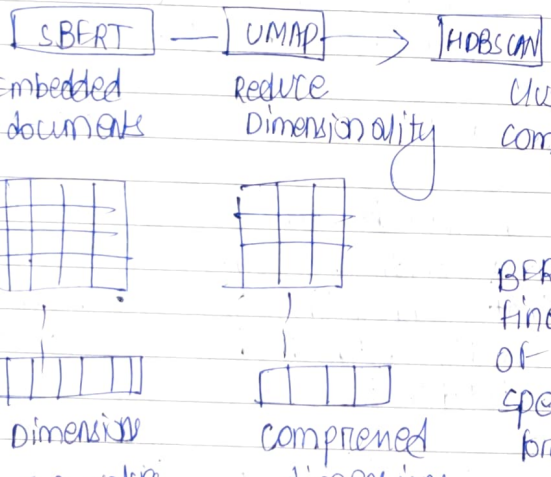
outlier

Ex - HDBSCAN

DBA are useful when we don't know no of clusters to be produced

Topic Modelling using BERTopic

1st stage -



BERTopic is a fine tuned varian of BERT is specifically train for topic model

In the 1st stage BERTopic uses SBERT in the embedding algo then it uses UMAP for reducing the dimension of the embedding

finally it uses a density based approach like HDBSCAN to create the clusters

In stage 2 BERTopic uses the one word embedd to count the frequency of each frequency which is called as term frequency

My cat is cute → 0 1 0 1 0 1 1
That is a cute dog → 1 1 1 1 1 0 0

that is a cute dog my cat
vocabulary of all i/p documents

Term frequency calculates the frequency of each term per document but to give a topic to a cluster it must calculate the frequency of each term per document per cluster

Therefore instead of TF, BERTopic calculates CTF (class term frequency) which is the frequency of each term in a document and in all documents across all ~~cluster~~ clusters

My cat is cute
That is a cute dog

class based term freq
1 2 1 2 1 1 1
4 6 3 1 8 0 5 0
4 1 3 2 0 3 0

that is a cute dog my

frequency of word w in cluster C = $||tf_{w,c}||$

The class term frequency gives the frequency of all the terms used somewhere in any document of the cluster. This also calculates the frequency of words like "the" "a" "is" which can be never be topic of cluster

Therefore BERTopic must eliminate words that are not useful. To do this BERTopic calculates the inverse document frequency for each term to put more weight on words that are meaningful and puts less weight on words that are common across all clusters.

ex - 'the', 'is', 'a'

$$IDF = \log\left(\frac{A}{cf_n} + 1\right)$$

A = Avg freq

cf_n = total freq of each word

$$= \log\left(\frac{29}{99 + 2291} + 1\right)$$

that is a cute dog! my cat.

After finding IDF, BERTopic tries to assign weight to each term in a cluster. the C-TF for a term is multiplied with its IDF value to finds its weight

$$\text{weight of term } w \text{ in a class} = \underbrace{||tf_{w,c}||}_{\text{CTF}} \times \underbrace{\log\left(\frac{A}{cf} + 1\right)}_{\text{IDF}}$$

After assigning weight to each term in a cluster BERTopic finds out topic which has highest weight

This term is choosen as the topic of the cluster

Final pipeline

Topic representation

clustering

SBERT → UMAP → HDBSCAN | count vectorizer → CTF IDF

putting the two steps together clustering and representing topics results in the full pipeline of BERTopic with this pipeline we can cluster semantically similar documents from the clusters generate topics represented by several keywords

The higher word's weight in a topic the more representative it is of that topic

One of the major advantages of BERTopic pipeline is it is modular as we can replace or change any unit without effect other units

Using generative models for Text clustering

Generative models like GPT can also be used for text clustering we have to provide the input documents & create a prompt for the generative model to cluster text. Along with the documents important keywords must also be provided

LLM

Keywords

[attention, text, nlp, transformer] + Documents

I have a topic that contains following documents
[DOCUMENTS]
The topic is described by following keywords
[keywords]
Based on the above info, give a short label of topic
Label

pretrained transformer model like nlp