University of East London
Pioneering Futures Since 1898

CN7050 – Intelligent Systems

Week 6: Language AI

How to Represent and Process Natural Language

Dr Shaheen Khatoon

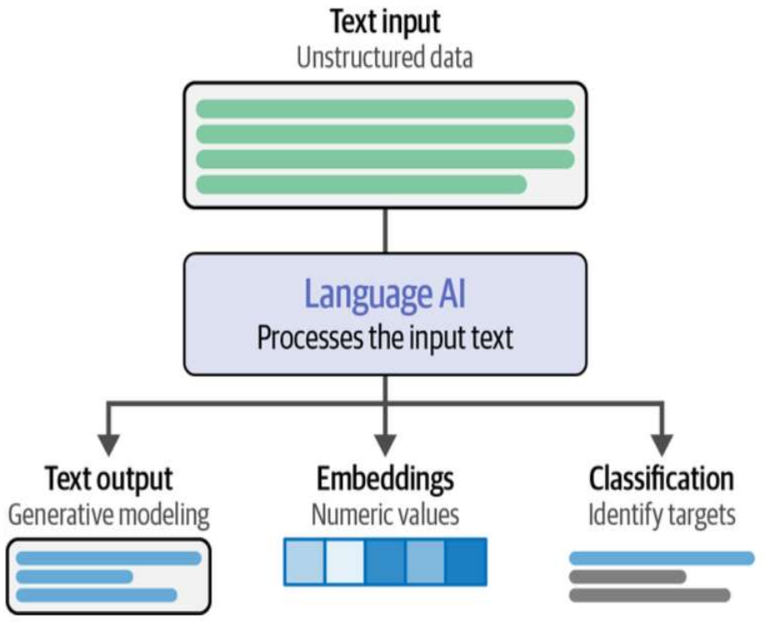
Office:EB.1.99
Email: s.khatoon@uel.ac.uk

Topics

- Early language representation models
- Developing Retrieval-based systems
- Text embedding Models
- Text Generation Models

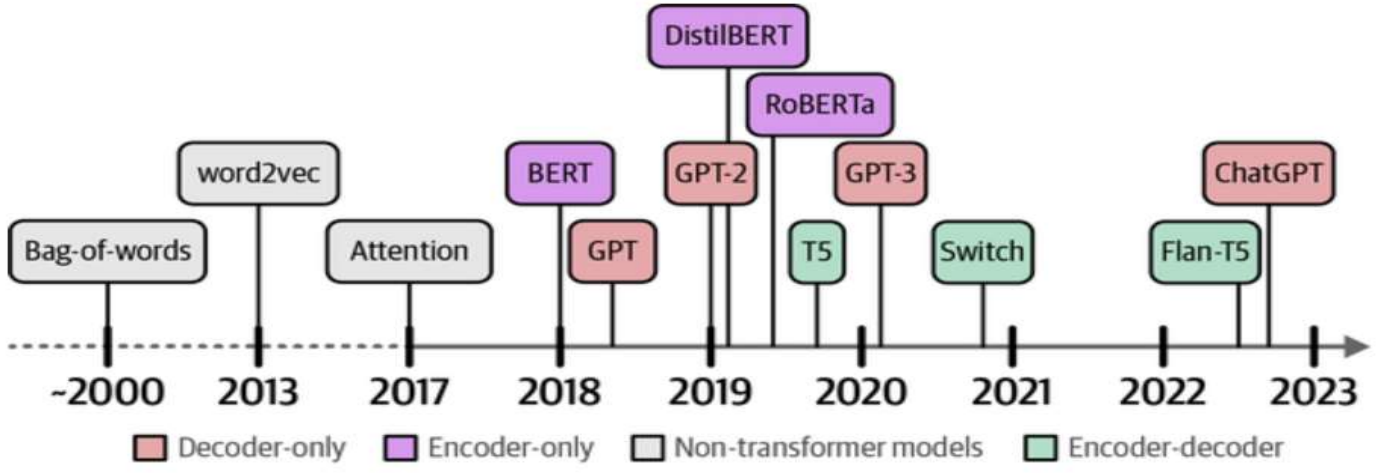
Representing Language for AI

- Text is unstructured data and cannot be directly fed to a machine in its natural
- Has to convert into a numerical representation that can be used by machine learning models.
- Loses its meaning when represented by zeros and ones (individual words).
- Throughout the history of **Language AI**, there has been a large focus on representing language in a structured manner so that it can more easily be used by computers.
- **Language AI** is capable of many tasks by processing textual input.



Representing Language for AI

- From 2012 onwards, developments in building AI systems (using deep neural networks) accelerated so that by the end of the decade, they yielded the first software system able to write articles indiscernible from those written by humans.
- The history of Language AI encompasses many developments and models aiming to represent and generate language



Representing Language for AI

- *Early Methods (Shallow Representations)*
  - **One-Hot Encoding:** Represents each word as a unique binary vector.
  - **Bag of Words (BoW):** Counts word occurrences, ignoring order and context.
  - **TF–IDF (Term Frequency–Inverse Document Frequency):** Weights words based on frequency and importance within documents.
- *Advanced Methods (Semantic Representations)*
  - **Word2Vec / GloVe:** Transforms words into **dense vectors of real numbers** capturing **semantic meaning** and relationships.
  - Enables operations like: *King – Man + Woman ≈ Queen*
- *Towards Large Language Models (LLMs)*
  - These foundational techniques underpin modern Large Language Models (LLMs) such as ChatGPT, which:
    - Learn context and meaning across massive text datasets.
    - Generate, summarize, and understand text in a human-like way.

Basic Terminology for Text Representation

- A **corpus** is known as an entire collection of text documents, for example,
  - a collection of emails, messages, or user reviews.
- **Document/Sentence:** a single document or sentence
- **Words**: set of words: ***(corpus) should be divided into fundamental units (words).***
- **Vocabulary:** Set of unique words in a corpus
- **Vector:** Numerical value of a word

Step Involved in Text Encoding

- The five major steps involved in handling text data for ML modeling are:
  - Reading the corpus
  - Tokenization
  - Cleaning/Stop word removal
  - Stemming
  - Converting into numerical form (vectors)

Tokenization

- The method of dividing the given text sequence/sentence or collection of words of a text document into **individual words**.
- It removes unnecessary characters such as punctuation.
- The final units post-tokenization are known as **tokens**.
- Let's say we have the following text:
  - **Input:** *He really liked the London City. He is there for two more days.*
  - **Tokenization** would result in the following tokens. We end up with 13 tokens for the input text:
  - *He, really, liked, the, London, City, He, is, there, for, two, more, days*

Stop word removal

- Tokens contain very common words such as “this,” “the,” “to,” “was,” “that,” etc.
- These words are known as **stopwords**, and they seem to add very little value to the analysis.
- If they are to be used in the analysis, it increases the computation overheads without adding values.
- Hence, it’s preferred to drop these stopwords from the tokens.
- We use **StopWordsRemover** to remove the stopwords:

Lemmatization

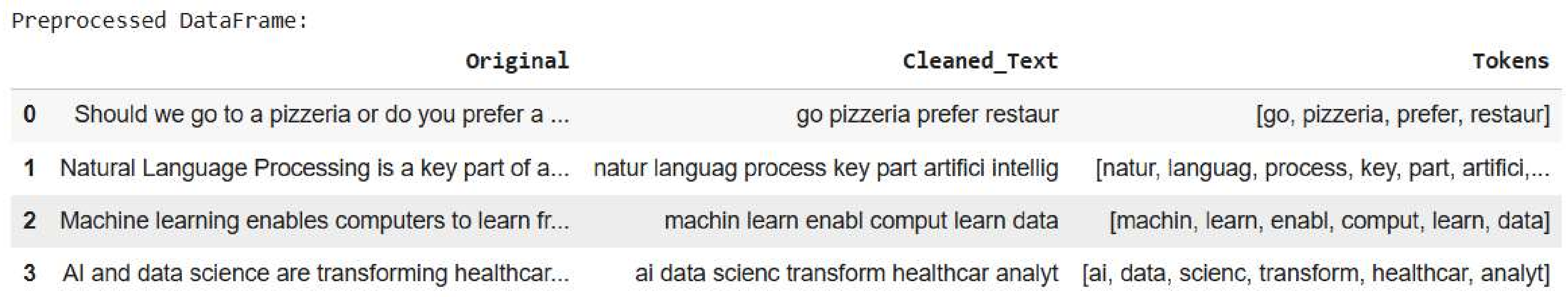
- Lemmatization reduces words to their base or dictionary form, known as the **lemma**.
- The base form is often referred to as the **root word or the canonical form**
- The main goal of lemmatization is to normalize words so that different **inflected forms of a word are treated as a single item**.
  - For example, the lemma of "running" would be "run," and the lemma of "played" would be "play."
- lemmatization helps to eliminate redundancy and improve the accuracy and efficiency of text analysis tasks.

Demo

- # Example Documents

```
documents = [
    "Should we go to a pizzeria or do you prefer a restaurant?",
    "Natural Language Processing is a key part of artificial intelligence.",
    "Machine learning enables computers to learn from data.",
    "AI and data science are transforming healthcare analytics."
]
```

After applying tokenization, removing stop words and lemmatization

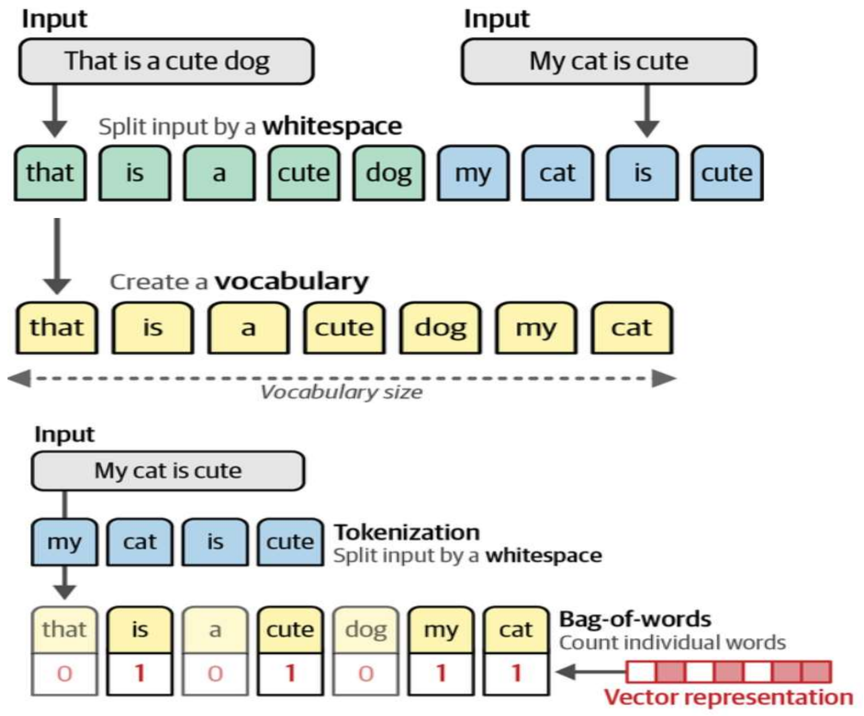


Preprocessed DataFrame:

	Original	Cleaned_Text	Tokens
0	Should we go to a pizzeria or do you prefer a ...	go pizzeria prefer restaur	[go, pizzeria, prefer, restaur]
1	Natural Language Processing is a key part of a...	natur languag process key part artifici intellig	[natur, languag, process, key, part, artifici,...
2	Machine learning enables computers to learn fr...	machin learn enabl comput learn data	[machin, learn, enabl, comput, learn, data]
3	AI and data science are transforming healthcar...	ai data scienc transform healthcar analyt	[ai, data, scienc, transform, healthcar, analyt]

Representing Text using One-hot encoding

- The first step is tokenization, the process of splitting up the sentences into individual words or subwords (tokens)
- after tokenization, combine all unique words from each sentence to create a vocabulary that we can use to represent the sentences.
- Using vocabulary, represent a document using 1s and 0s showing presence or absence of vocabulary word in a document
- Finally create document vector

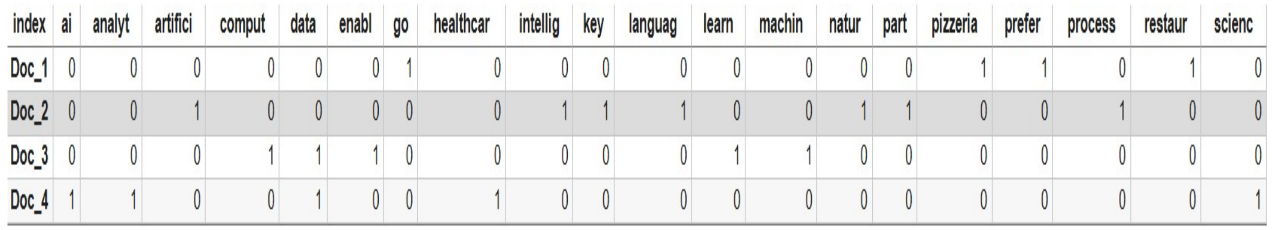


Representing Text using One-hot encoding

```
documents = [
    "Should we go to a pizzeria or do you prefer a restaurant?",
    "Natural Language Processing is a key part of artificial
intelligence.",
    "Machine learning enables computers to learn from data.",
    "AI and data science are transforming healthcare analytics."
]
```

Vocabulary: ['ai', 'analyt', 'artifici', 'comput', 'data', 'enabl', 'go', 'healthcar', 'intellig', 'key', 'languag', 'learn', 'machin', 'natur', 'part', 'pizzeria', 'prefer', 'process', 'restaur', 'scienc', 'transform']

One-Hot Encoding Matrix for All Documents:



index	ai	analyt	artifici	comput	data	enabl	go	healthcar	intellig	key	languag	learn	machin	natur	part	pizzeria	prefer	process	restaur	scienc
Doc_1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	0	1	0
Doc_2	0	0	1	0	0	0	0	0	1	1	1	0	0	1	1	0	0	1	0	0
Doc_3	0	0	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0	0	0	0
Doc_4	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1

Each document is represented by a **binary vector $\in \{0,1\}^{|V|}$**

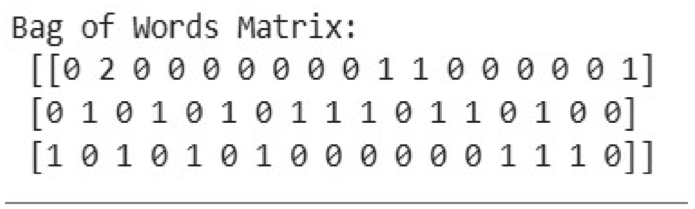
Representing Text using One-hot encoding

- Advantages:
  - Simple to implement
- Disadvantages
  - No ranking
  - Sparse matrix
  - Out-of-vocabulary words can't be accepted
  - Ignore semantic meaning and word order
- Basis for advanced models

Representing text using - Bag-of-words

Represents documents as a frequency count of words
Each document is a **count vector in $\mathbb{N}^{|V|}$**

```
# Example of usage
corpus =
["This movie is awesome awesome?",
"I do not say is good, but neither awesome",
 "Awesome? Only a fool can say that"]


 Vocabulary: ['a', 'awesome', 'awesome?', 'but',
 'can', 'do', 'fool', 'good,', 'i', 'is',
 'movie', 'neither', 'not', 'only', 'say',
 'that', 'this']
```



```
Bag of Words Matrix:
 [[0 2 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1]
 [0 1 0 1 0 1 0 1 1 1 0 1 1 0 1 0 0]
 [1 0 1 0 1 0 1 0 0 0 0 0 0 1 1 1 0]]
```

- classic method, it is by no means completely obsolete. used to complement more recent language models.
- Advantages:
  - Simple to implement
  - Ranked document based on term frequency
- Disadvantages
  - feature matrix will be a huge sparse matrix
  - Out-of-vocabulary words cant be accepted
  - Ignore semantic meaning/context and word order
  - *For example, the how word "**awesome**" is associated with a review with a positive, neutral, or negative meaning. Without context, the frequency of the word "awesome" alone does not tell us the sentiment of the review.*

Representing Text using TF-IDF

- **Term Frequency (TF)?**
  - Terms frequency is not necessarily the best representation for text.
  - A word appearing **100 times** isn't **100× more important** than one appearing once.
  - So, instead of raw frequency, we take the **logarithm (base 10)** of the frequency.
    $TF=log_{10}(1+frequency)$
  - Words **absent in a document** get a score of **0**
- **Inverse Document Frequency (IDF)**
  - Some words appear in **almost every document** (e.g., "the", "is", "and") — they add **little meaning**.
  - We want to emphasize **rare but meaningful words**.
  - **IDF** gives higher weight to words that appear in **fewer documents**: $\mathrm{idf}_t = \log_{10}(N/\mathrm{df}_t)$

  Where, N = total number of documents and Df_t number of documents containing the term t
- The tf-idf weight of a term is the product of its tf weight and its idf weight.

$$tf.idf = \log_{10}(1 + \mathrm{tf}_{t,\mathrm{d}}) \text{ x} \log_{10}\left(\frac{N}{\mathrm{df}_t}\right)$$

Representing Text using TF-IDF

- Use the same corpus

```
# Example of usage
corpus =
["This movie is awesome awesome?",
"I do not say is good, but neither awesome",
 "Awesome? Only a fool can say that"]
```

```
Vocabulary: ['a', 'awesome', 'awesome?', 'but', 'can', 'do', 'fool', 'good,', 'i', 'is', 'movie', 'neither', 'not', 'only', 'say', 'that', 'this']
```

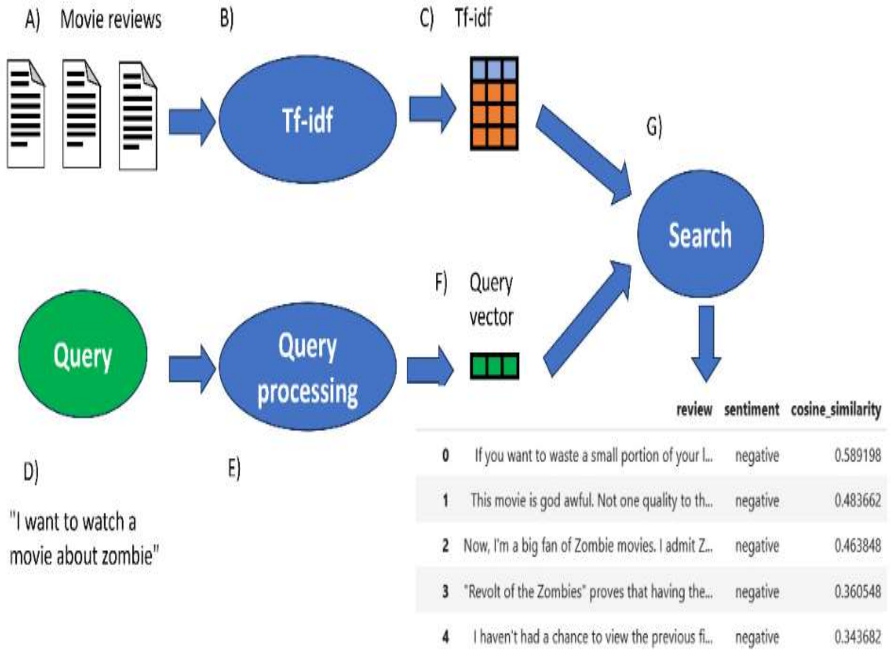
```
TF-IDF Matrix:
 [[0.         0.4        0.         0.         0.         0.
  0.         0.         0.         0.2        0.28109303 0.
  0.         0.         0.         0.         0.28109303]
 [0.         0.11111111 0.         0.1561628  0.         0.1561628
  0.         0.1561628  0.1561628  0.11111111 0.         0.1561628
  0.1561628  0.         0.11111111 0.         0.         ]
 [0.20078073 0.         0.20078073 0.         0.20078073 0.
  0.20078073 0.         0.         0.         0.         0.
  0.         0.20078073 0.14285715 0.20078073 0.         ]]
```

Note how word frequencies changed after this normalization.

Use of tf-idf in Information Retrieval systems

Powerful for Information Retrieval: TF-IDF efficiently identifies *important keywords* in documents — still widely used in **search engines**, and **document ranking**.

- the system uses a search in a vector space;
- This can be bag-of-words or the TF-IDF
  - For example, take a set of documents and calculate the TF-IDF.
  - Calculate tf-idf for query
  - Calculate a score (usually cosine similarity) between each of the documents and conduct rank based on the score.

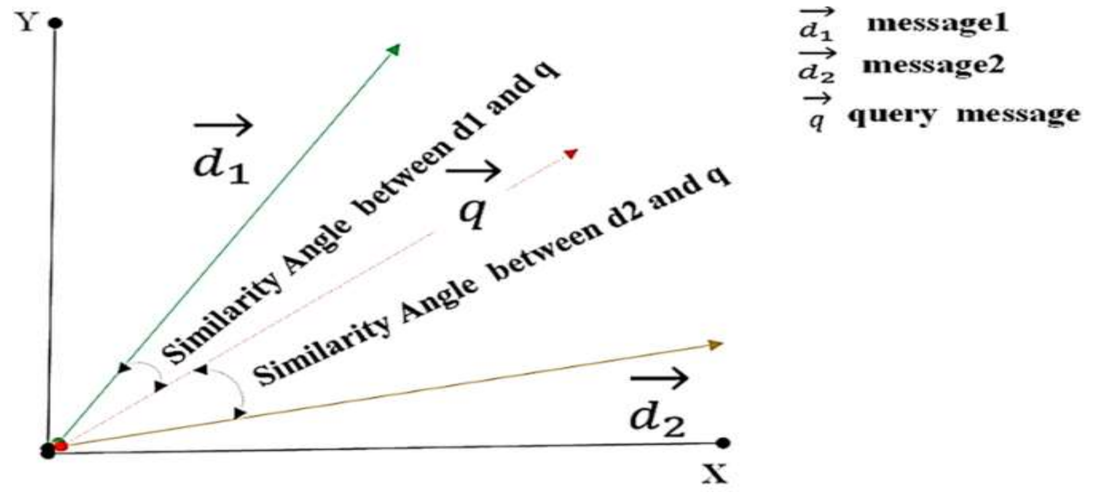


	review	sentiment	cosine_similarity
0	If you want to waste a small portion of your l...	negative	0.589198
1	This movie is god awful. Not one quality to th...	negative	0.483662
2	Now, I'm a big fan of Zombie movies. I admit Z...	negative	0.463848
3	"Revolt of the Zombies" proves that having the...	negative	0.360548
4	I haven't had a chance to view the previous fi...	negative	0.343682

Cosine Similarity

- For a document *d* and a *query q* in *vector space*, use the following formula:

$$(\boldsymbol{Cos}\,\boldsymbol{q}\ ,\mathbf{d}) = \frac{\boldsymbol{q}.\boldsymbol{d}}{|\boldsymbol{q}||\boldsymbol{d}|}$$

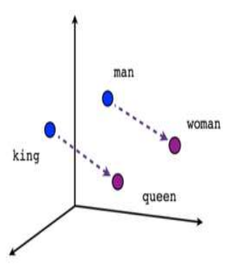


Better Text Representation with Word Embeddings

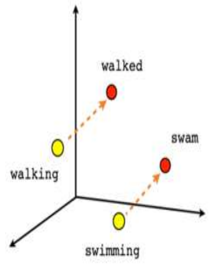
- Bag-of-words, although an elegant approach, has a flaw.
  - It considers language to be nothing more than an almost **literal bag of words**
  - ignores *word order*, c*ontext*, and *semantic meaning*
  - Serve a foundation for advanced language representation techniques
- Word2Vec (2013)
  - A breakthrough in representing meaning and relationships between words.
  - Learn **Embedding**- dense vector representations that capture:
    - context of words,
    - Relationships among them
    - Semantic meanings
- Core Idea
  - Words with similar meanings are mapped to similar embeddings.
  - Cosine similarity measures how close two words are in meaning.
- Why it Matters
  - Enables semantic similarity, analogy reasoning, and context-aware NLP.
  - Forms the foundation of modern NLP — powering models like **Word2Vec, GloVe, BERT, and GPT.**

Word Embeddings

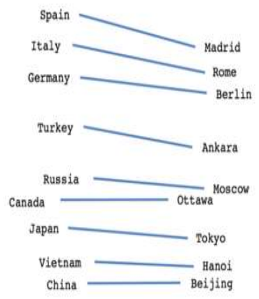
- Converts words into dense vectors of real numbers.
- Words appearing in similar contexts have similar embeddings.



Male-Female



Verb tense



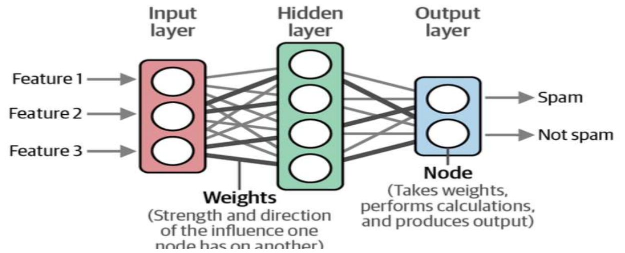
Country-Capital

Closer points are closer in meaning and they form clusters …

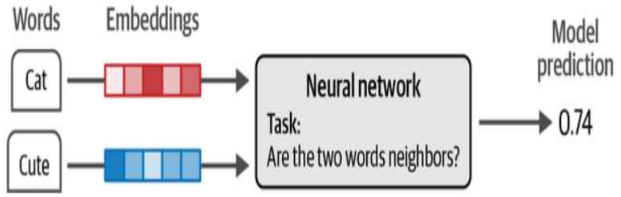
- vector[Queen] ≈ vector[King] - vector[Man] + vector[Woman]
- vector[Paris] ≈ vector[France] - vector[Italy] + vector[Rome]
  - This can be interpreted as “France is to Paris as Italy is to Rome”.

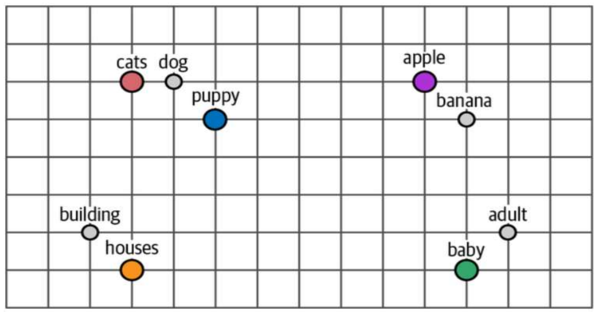
Word2Vec

- Learns semantic representations of words by training on vast amounts of textual data, like the entirety of Wikipedia.
- Uses neural networks, consist of interconnected layers of nodes that process information.
- Generates word embeddings by looking at which other words they tend to appear next to in a given sentence.
  - Start by assigning every word in vocabulary with a **vector embedding**, *say of 50 values for each word initialized with random values.*
  - Then in every training step, it take pairs of words from the **training data** and a model attempts to predict whether or not they are likely to be **neighbors in a sentence.**
  - During this training process, it learns the relationship between words and distills that information into the embedding.
  - If the two words tend to have the same neighbors, their embeddings will be closer to one another and vice versa.

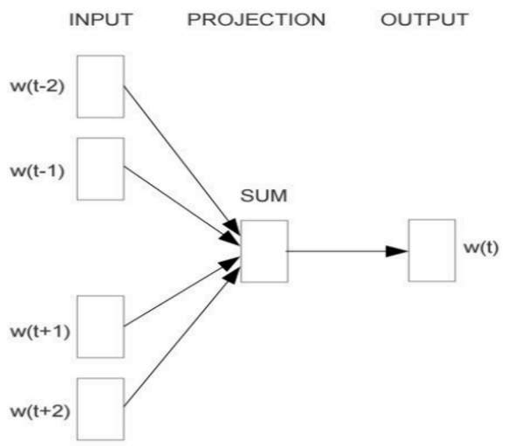


Input layer
Hidden layer
Output layer
Feature 1
Feature 2
Feature 3
Spam
Not spam
Weights
(Strength and direction of the influence one node has on another)
Node
(Takes weights, performs calculations, and produces output)

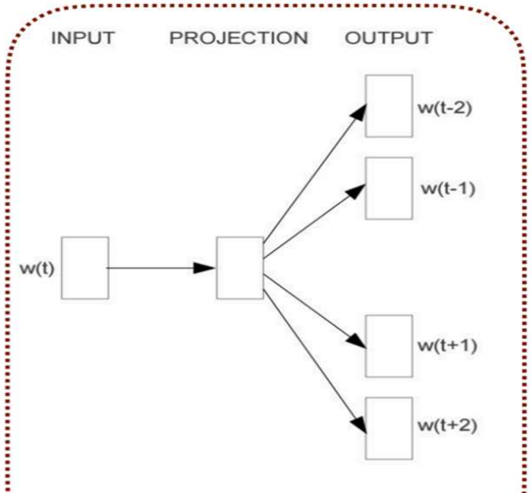




Word2vec: Two ways



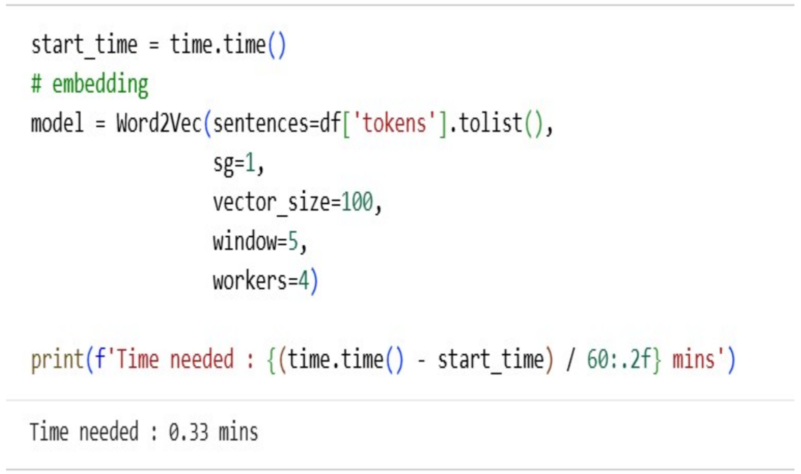
Continuous Bag of Words (CBOW)



Skip-grams

Word2vec: Demo

trains a Word2Vec model from tokenized text data: See lab tutorial



```
start_time = time.time()
# embedding
model = Word2Vec(sentences=df['tokens'].tolist(),
                 sg=1,
                 vector_size=100,
                 window=5,
                 workers=4)

print(f'Time needed : {(time.time() - start_time) / 60:.2f} mins')
```

```
Time needed : 0.33 mins
```

sentences=df['tokens'].tolist(): training corpus

sg=1:

This parameter chooses the **training algorithm**:

- sg=0: CBOW **(Continuous Bag of Words)** — predicts a target word from surrounding context words.
- sg=1: **Skip-gram** — predicts the surrounding context words given a target word.

Skip-gram (sg=1) generally performs better for **rare words** and captures **semantic relationships** more effectively.

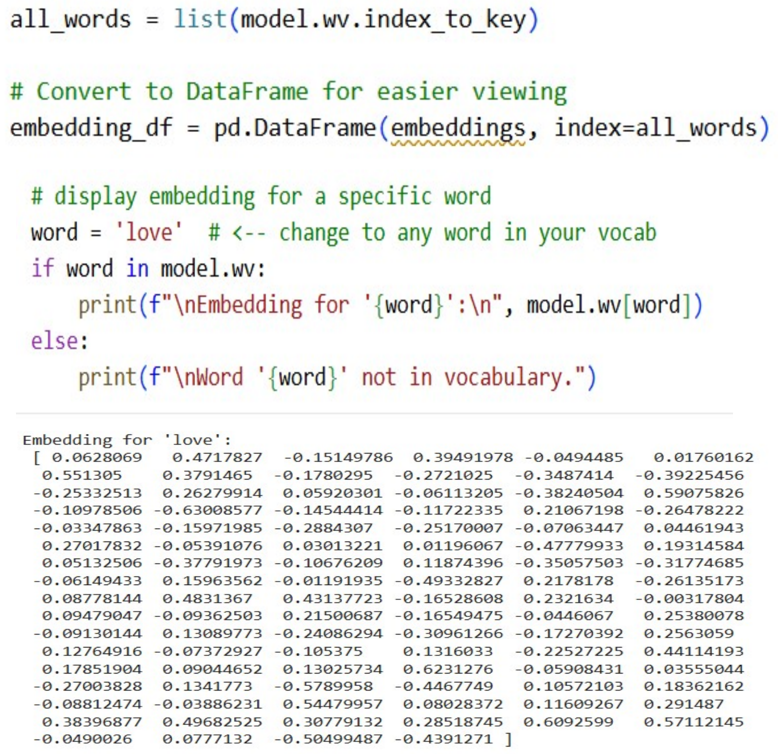
vector_size=100 : set the embedding size, Each word will be represented as a 100-dimensional vector.

window=5 : Defines the **context window size** — how many words to the left and right of a target word are considered during training.

workers=4: Number of CPU cores to use for parallel processing

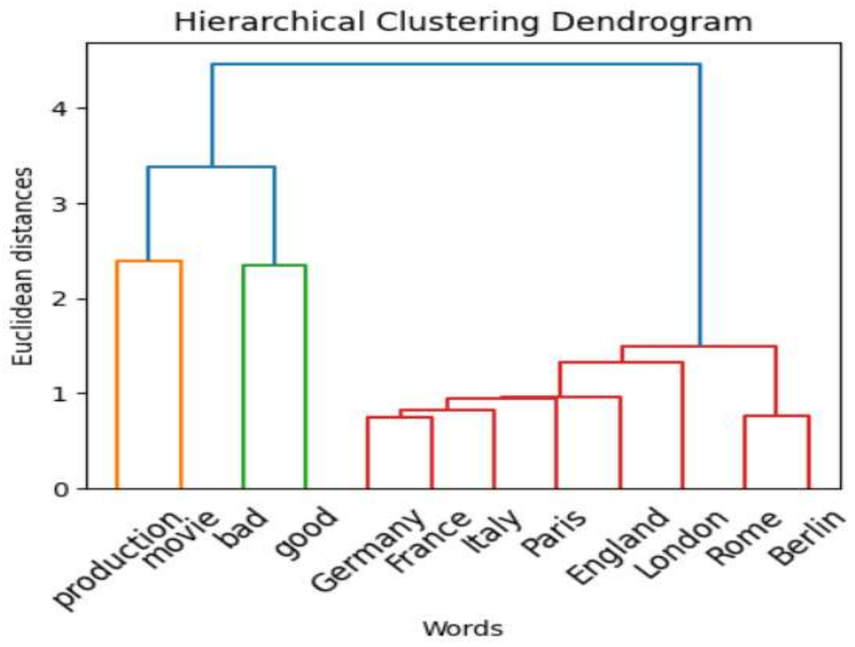
Word2vec: Demo

Print Embeddings



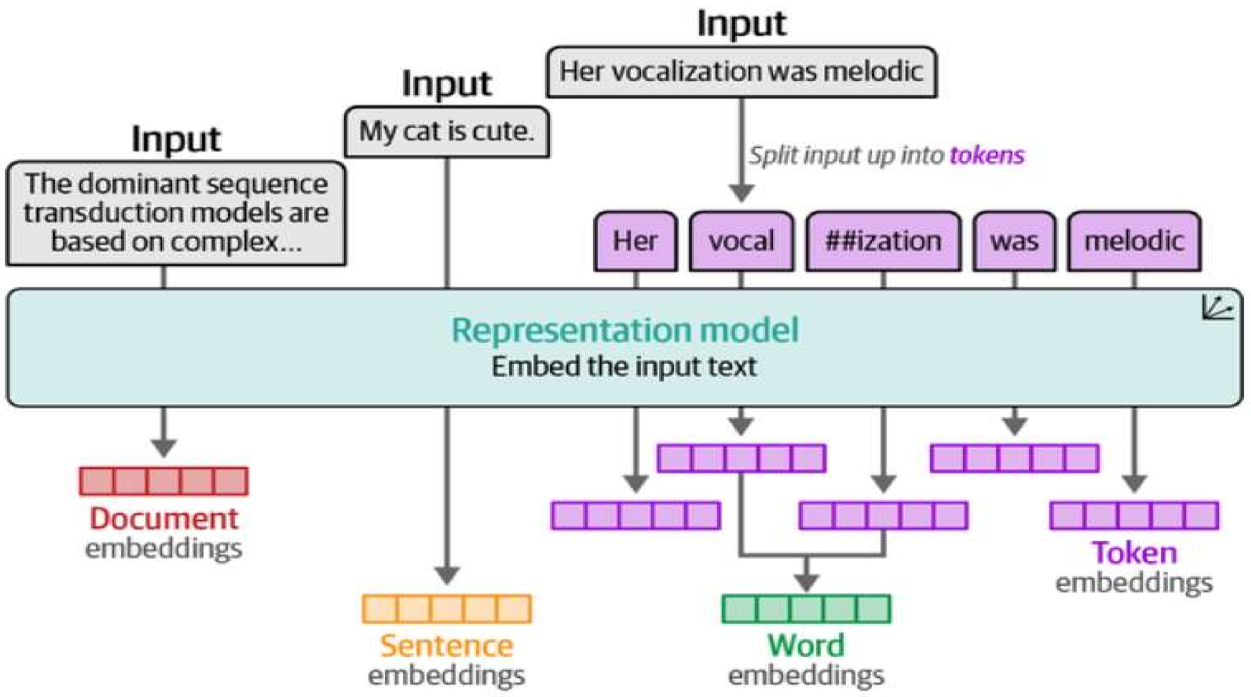
```python
all_words = list(model.wv.index_to_key)

# Convert to DataFrame for easier viewing
embedding_df = pd.DataFrame(embeddings, index=all_words)

 # display embedding for a specific word
 word = 'love'  # <-- change to any word in your vocab
 if word in model.wv:
     print(f"\nEmbedding for '{word}':\n", model.wv[word])
 else:
     print(f"\nWord '{word}' not in vocabulary.")
```

```
Embedding for 'love':
 [ 0.0628069   0.4717827  -0.15149786  0.39491978 -0.0494485   0.01760162
  0.551305    0.3791465  -0.1780295  -0.2721025  -0.3487414  -0.39225456
 -0.25332513  0.26279914  0.05920301 -0.06113205 -0.38240504  0.59075826
 -0.10978506 -0.63008577 -0.14544414 -0.11722335  0.21067198 -0.26478222
 -0.03347863 -0.15971985 -0.2884307  -0.25170007 -0.07063447  0.04461943
  0.27017832 -0.05391076  0.03013221  0.01196067 -0.47779933  0.19314584
  0.05132506 -0.37791973 -0.10676209  0.11874396 -0.35057503 -0.31774685
 -0.06149433  0.15963562 -0.01191935 -0.49332827  0.2178178  -0.26135173
  0.08778144  0.4831367   0.43137723 -0.16528608  0.2321634  -0.00317804
  0.09479047 -0.09362503  0.21500687 -0.16549475 -0.0446067   0.25380078
 -0.09130144  0.13089773 -0.24086294 -0.30961266 -0.17270392  0.2563059
  0.12764916 -0.07372927 -0.105375    0.1316033  -0.22527225  0.44114193
  0.17851904  0.09044652  0.13025734  0.6231276  -0.05908431  0.03555044
 -0.27003828  0.1341773  -0.5789958  -0.4467749   0.10572103  0.18362162
 -0.08812474 -0.03886231  0.54479957  0.08028372  0.11609267  0.291487
  0.38396877  0.49682525  0.30779132  0.28518745  0.6092599   0.57112145
 -0.0490026   0.0777132  -0.50499487 -0.4391271 ]
```



words that have similar meanings should be closer together:

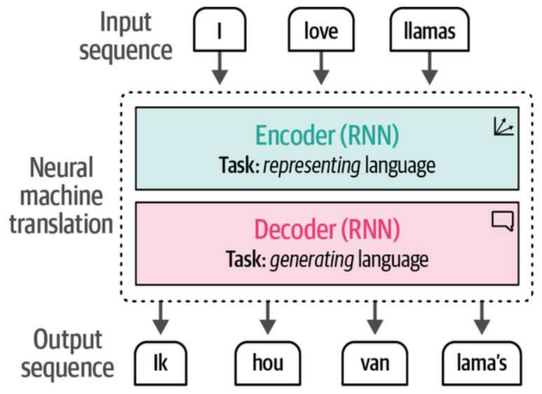
Types of Embeddings



26

Encoding and Decoding Context with Attention

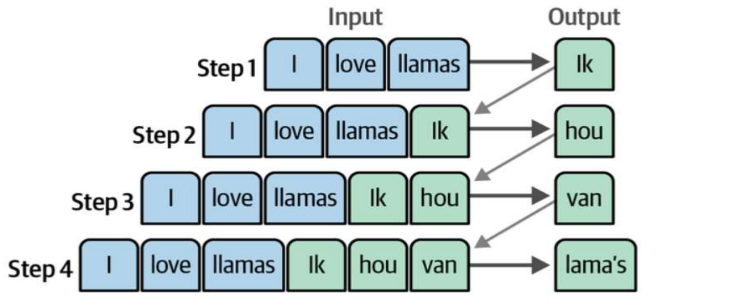
- word2vec creates static representations of words.
  - For instance, the word “bank” will always have the same embedding regardless of the context in which it is used.
  - However, “bank” can refer to both a financial bank as well as the bank of a river.
  - Its meaning, and therefore its embeddings, should change depending on the context.
- **Recurrent neural networks (RNNs).**
  - These are variants of neural networks that can model sequences as an additional input, used for two tasks:
  - **encoding or representing an input sentence**
  - **decoding or generating an output sentence.**

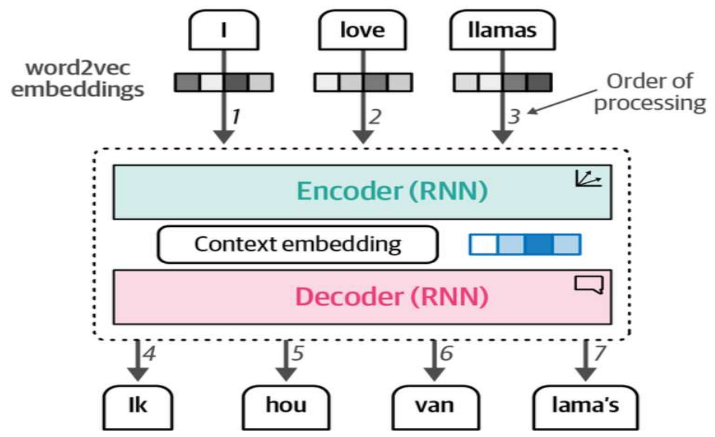


Two recurrent neural networks (decoder and encoder) translating an input sequence from English to Dutch.

Encoding and Decoding Context with Attention

- Each step in this architecture is ***autoregressive***.
- When generating the next word, ach previous output token is used as input to generate the next token.
- The encoding step aims to represent the input as well as generating the **context in the form of an embedding**, which serves as the input for the decoder.
- To generate this representation, it takes embeddings as its inputs for words, which means we can use word2vec for the initial representations.

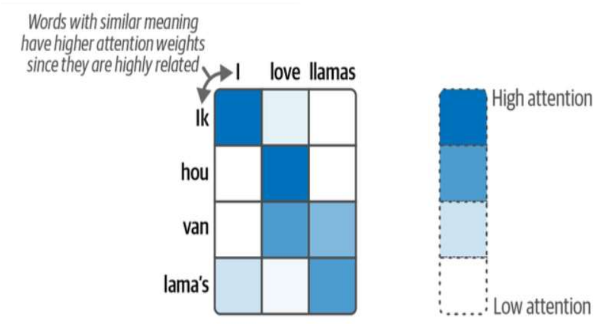




28

Encoding and Decoding Context with Attention

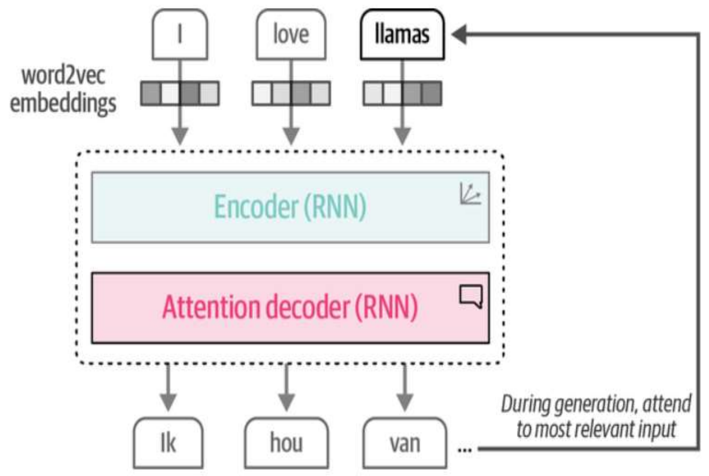
- **Context embedding**, makes it difficult to deal with longer sentences since it is merely a single embedding representing the entire input.
- In 2014, a solution called ***attention*** was introduced that highly improved upon the original architecture.
- **Attention** allows a model to focus on **parts of the input sequence that are relevant to one another** ("attend" to each other) and amplify their signal
- Attention selectively determines which words are most important in a given sentence.
- For instance, the output word "lama's" is Dutch for "llamas," which is why the attention between both is high. Similarly, the words "lama's" and "I" have lower attention since they aren't as related.



Encoding and Decoding Context with Attention

- By adding these **attention mechanisms to the decoder step,** the RNN can generate signals for each input word in the sequence related to the potential output.

- Instead of passing only a **context embedding** to the decoder, the **hidden states of all input words are passed.**



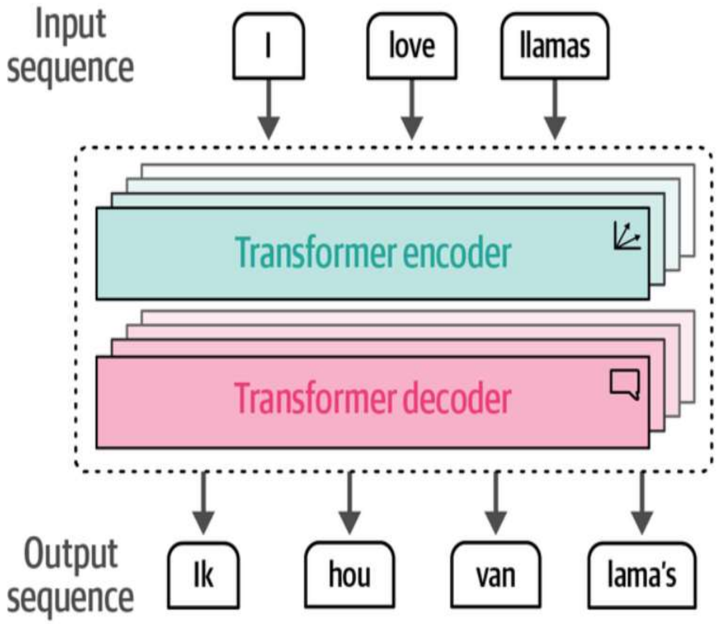
30

Transformer: Attention Is All You Need

- The **true power of attention**, and what drives the amazing abilities of large language models, was first explored in the well-known *"Attention is all you need" paper released in 2017.*

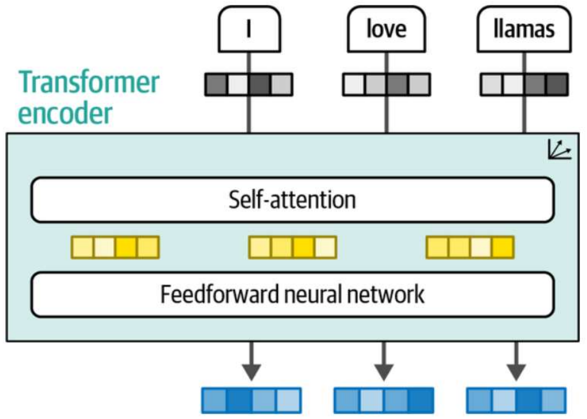
- The authors proposed a network architecture called the **Transformer**, which was solely based on the **attention mechanism** and removed the recurrence network

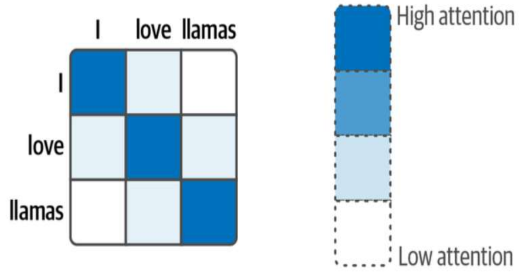
- Compared to the recurrence network, the Transformer could be trained in parallel, which tremendously sped up training.



Transformer: Encoder

- The encoder block in the Transformer consists of two parts,
  - **self-attention**
  - **feedforward neural network,**
- Compared to previous methods of attention, self-attention can attend to different positions within a single sequence
- Instead of processing one token at a time, it can be used to look at the entire sequence in one go.



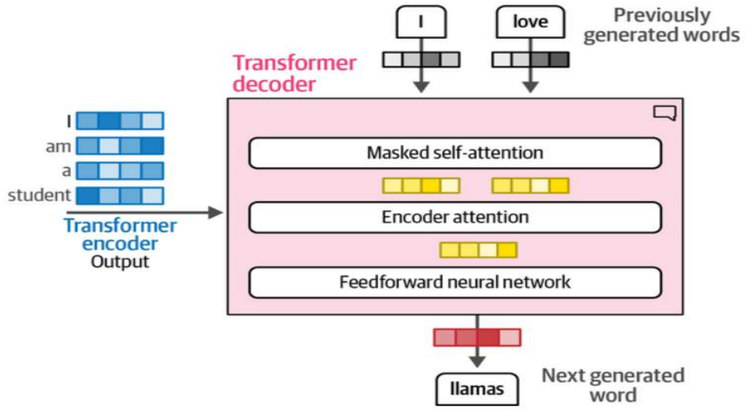


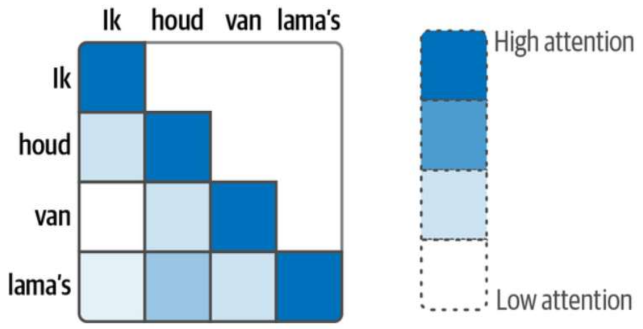
Transformer: Decoder

- decoder has an additional layer that pays attention to the output of the encoder (to find the relevant parts of the input).

- the self-attention layer in the decoder **masks future positions** so it only attends to earlier positions to prevent leaking information when generating the output.

- these building blocks create the Transformer architecture and are the foundation of many impactful models in Language AI, such as BERT and GPT





BERT: Encoder-Only Models

- The original Transformer model is an encoder-decoder architecture that serves translation tasks well but cannot easily be used for other tasks, like text classification.

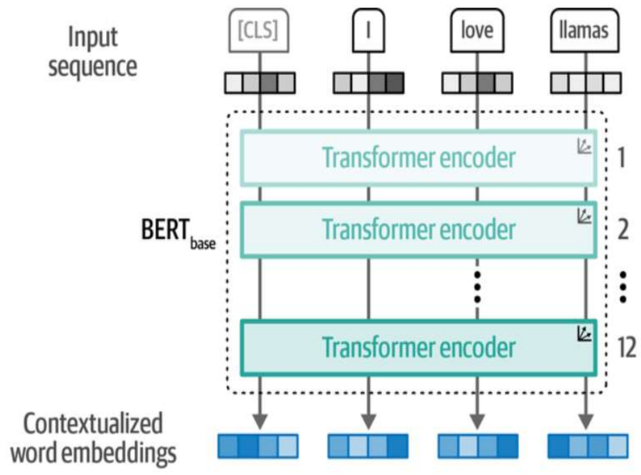
- In 2018, a new architecture called **Bidirectional Encoder Representations from Transformers (BERT)** was introduced that could be leveraged for a wide variety of tasks and would serve as the foundation of Language AI for years to come.

-

- BERT is an encoder-only architecture that focuses **on representing language**,

- This means that it only uses the encoder and removes the decoder entirely.

The architecture of a BERT base model with 12 encoders.



[CLS] or classification token is used as the representation for the entire input. Use as input embedding for fine-tuning the model on specific tasks, like classification.

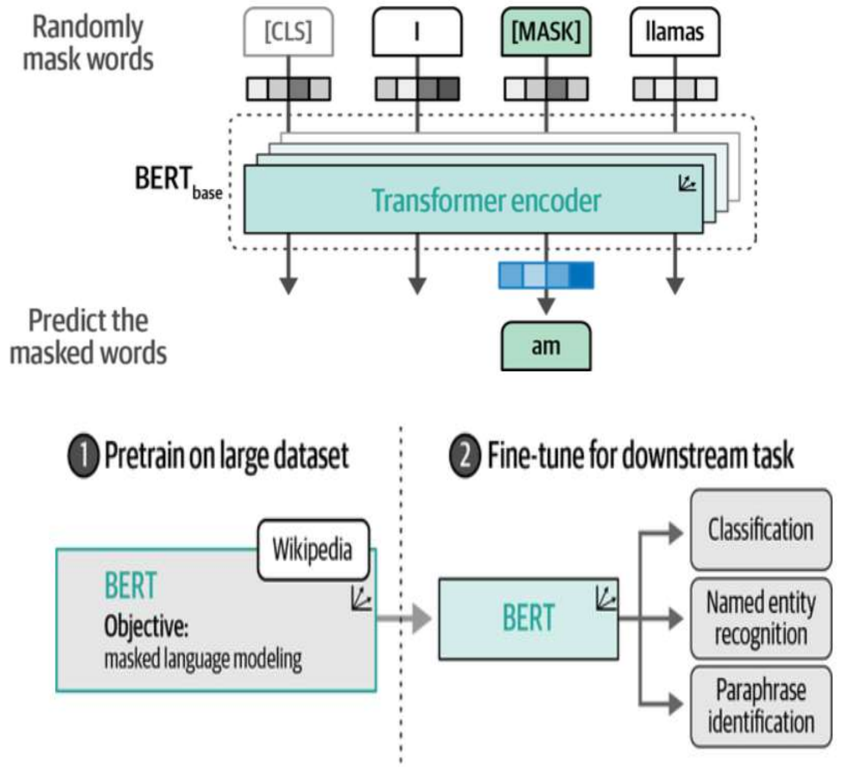
Pretrained models

- Since training a model requires huge datasets of text and significan computation, researchers often use common pretrained models
- Examples
  - Google's BERT model
  - Huggingface's various Transformer models
  - OpenAI's and GPT-3 models

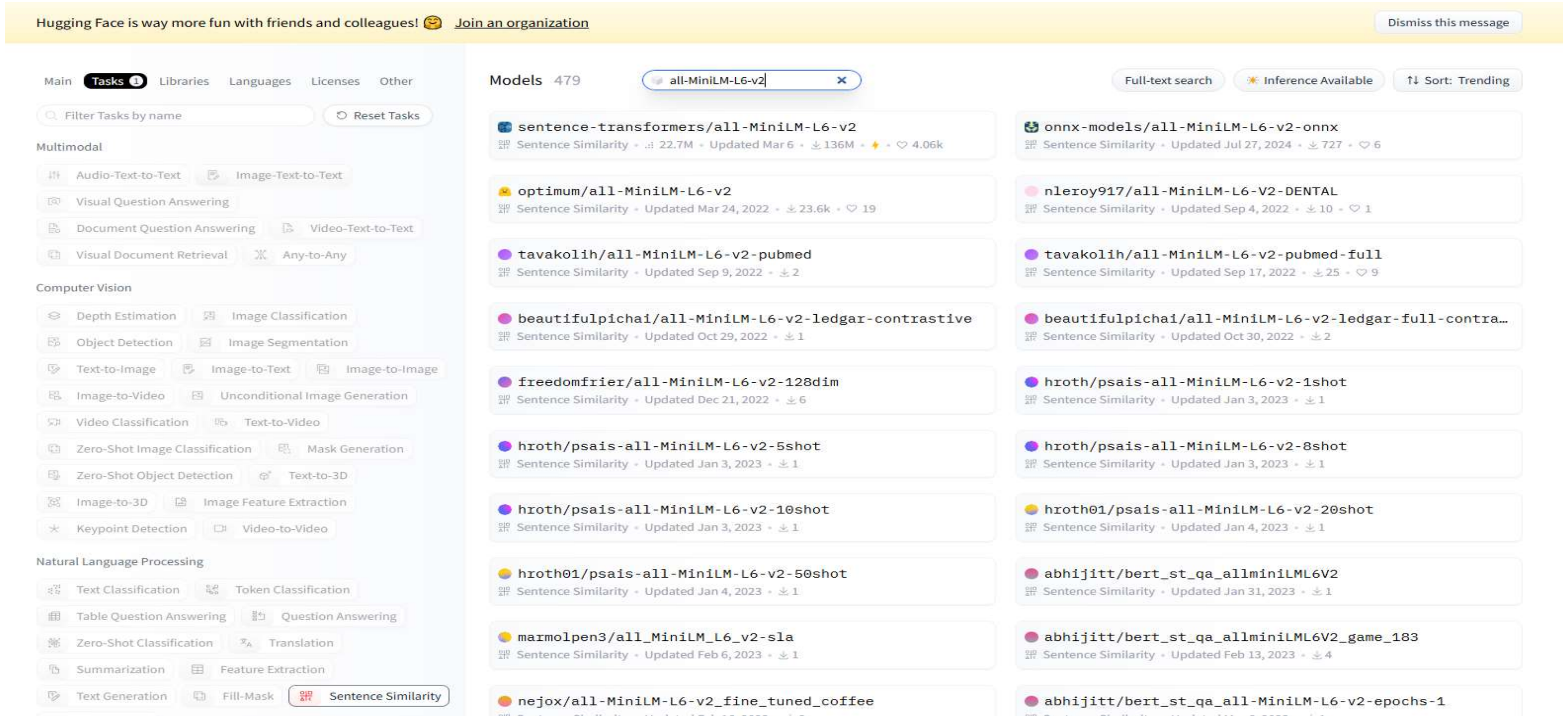
35

BERT: Encoder-Only Models

- Training uses **masked language modelling**, which masks a part of the input for the model to predict.
- This prediction task is difficult but allows BERT to create more accurate (intermediate) representations of the input.
- This architecture and training procedure makes BERT and related architectures incredible at representing contextual language
- BERT-like models are commonly used for **transfer learning**, which involves first pretraining it for language modeling and then fine-tuning it for a specific task.



Huggingface transformer Models



Hugging Face is way more fun with friends and colleagues! 🤗 Join an organization

Dismiss this message

Models 479 all-MiniLM-L6-v2

sentence-transformers/all-MiniLM-L6-v2
Sentence Similarity • 22.7M • Updated Mar 6 • 136M • • 4.06k

onnx-models/all-MiniLM-L6-v2-onnx
Sentence Similarity • Updated Jul 27, 2024 • 727 • 6

optimum/all-MiniLM-L6-v2
Sentence Similarity • Updated Mar 24, 2022 • 23.6k • 19

nleroy917/all-MiniLM-L6-V2-DENTAL
Sentence Similarity • Updated Sep 4, 2022 • 10 • 1

tavakolih/all-MiniLM-L6-v2-pubmed
Sentence Similarity • Updated Sep 9, 2022 • 2

tavakolih/all-MiniLM-L6-v2-pubmed-full
Sentence Similarity • Updated Sep 17, 2022 • 25 • 9

beautifulpichai/all-MiniLM-L6-v2-ledgar-contrastive
Sentence Similarity • Updated Oct 29, 2022 • 1

beautifulpichai/all-MiniLM-L6-v2-ledgar-full-contra…
Sentence Similarity • Updated Oct 30, 2022 • 2

freedomfrier/all-MiniLM-L6-v2-128dim
Sentence Similarity • Updated Dec 21, 2022 • 6

hroth/psais-all-MiniLM-L6-v2-1shot
Sentence Similarity • Updated Jan 3, 2023 • 1

hroth/psais-all-MiniLM-L6-v2-5shot
Sentence Similarity • Updated Jan 3, 2023 • 1

hroth/psais-all-MiniLM-L6-v2-8shot
Sentence Similarity • Updated Jan 3, 2023 • 1

hroth/psais-all-MiniLM-L6-v2-10shot
Sentence Similarity • Updated Jan 3, 2023 • 1

hroth01/psais-all-MiniLM-L6-v2-20shot
Sentence Similarity • Updated Jan 4, 2023 • 1

hroth01/psais-all-MiniLM-L6-v2-50shot
Sentence Similarity • Updated Jan 4, 2023 • 1

abhijitt/bert_st_qa_allminiLML6V2
Sentence Similarity • Updated Jan 31, 2023 • 1

marmolpen3/all_MiniLM_L6_v2-sla
Sentence Similarity • Updated Feb 6, 2023 • 1

abhijitt/bert_st_qa_allminiLML6V2_game_183
Sentence Similarity • Updated Feb 13, 2023 • 4

nejox/all-MiniLM-L6-v2_fine_tuned_coffee

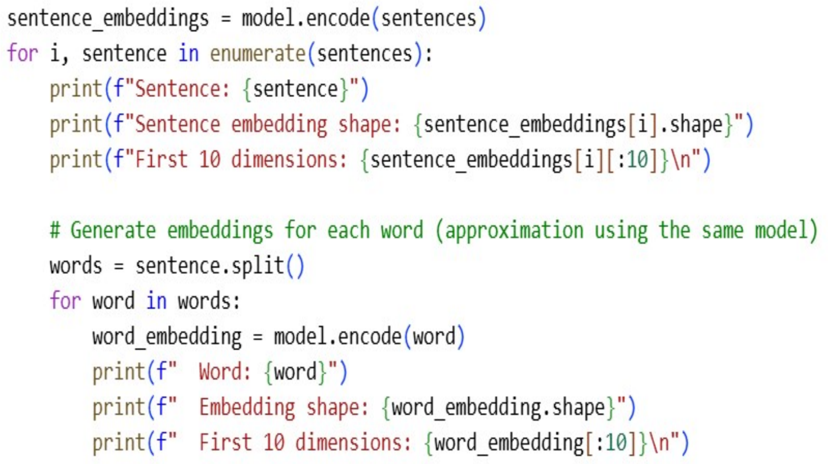
abhijitt/bert_st_qa_all-MiniLM-L6-v2-epochs-1

https://huggingface.co/models?pipeline_tag=sentence-similarity&sort=trending&search=all-MiniLM-L6-v2

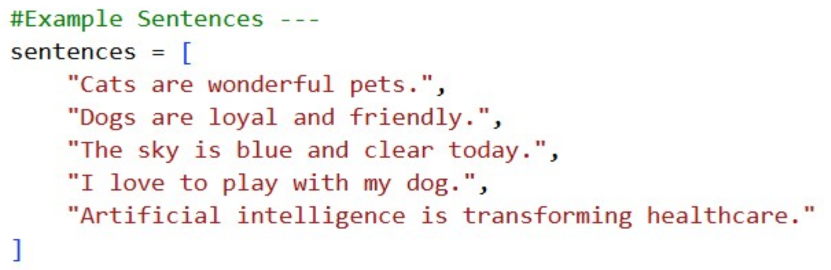
37

Use of Sentence Transformer Model

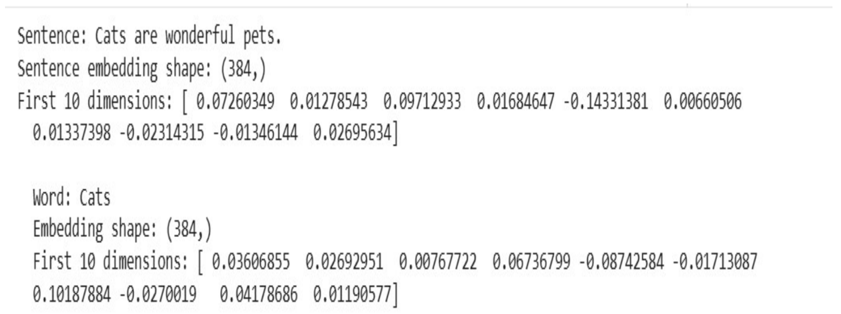
```
!pip install -q sentence-transformers

from sentence_transformers import
SentenceTransformer

model = SentenceTransformer('all-MiniLM-L6-
v2')
```



```
sentence_embeddings = model.encode(sentences)
for i, sentence in enumerate(sentences):
    print(f"Sentence: {sentence}")
    print(f"Sentence embedding shape: {sentence_embeddings[i].shape}")
    print(f"First 10 dimensions: {sentence_embeddings[i][:10]}\n")

    # Generate embeddings for each word (approximation using the same model)
    words = sentence.split()
    for word in words:
        word_embedding = model.encode(word)
        print(f"  Word: {word}")
        print(f"  Embedding shape: {word_embedding.shape}")
        print(f"  First 10 dimensions: {word_embedding[:10]}\n")
```

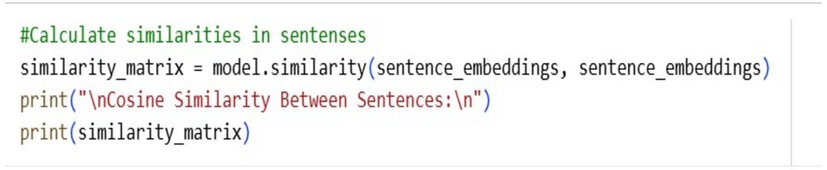


```
#Example Sentences ---
sentences = [
    "Cats are wonderful pets.",
    "Dogs are loyal and friendly.",
    "The sky is blue and clear today.",
    "I love to play with my dog.",
    "Artificial intelligence is transforming healthcare."
]
```

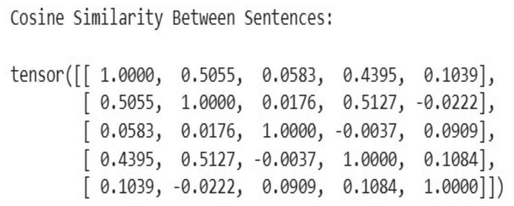


```
Sentence: Cats are wonderful pets.
Sentence embedding shape: (384,)
First 10 dimensions: [ 0.07260349  0.01278543  0.09712933  0.01684647 -0.14331381  0.00660506
  0.01337398 -0.02314315 -0.01346144  0.02695634]

  Word: Cats
  Embedding shape: (384,)
  First 10 dimensions: [ 0.03606855  0.02692951  0.00767722  0.06736799 -0.08742584 -0.01713087
  0.10187884 -0.0270019   0.04178686  0.01190577]
```

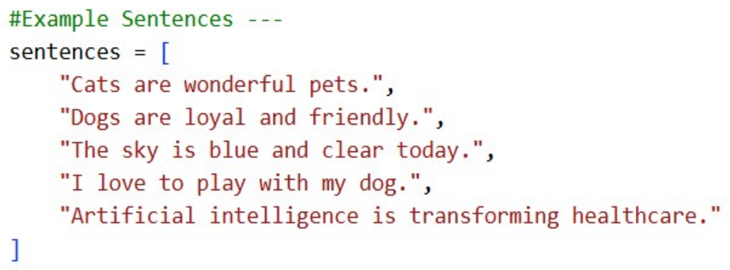
Use of Sentence Transformer Model



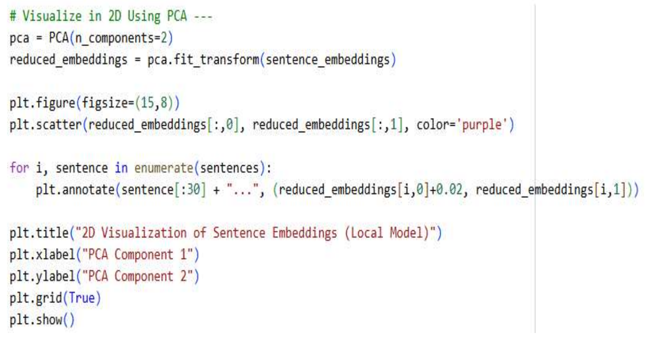
```
#Calculate similarities in sentenses
similarity_matrix = model.similarity(sentence_embeddings, sentence_embeddings)
print("\nCosine Similarity Between Sentences:\n")
print(similarity_matrix)
```



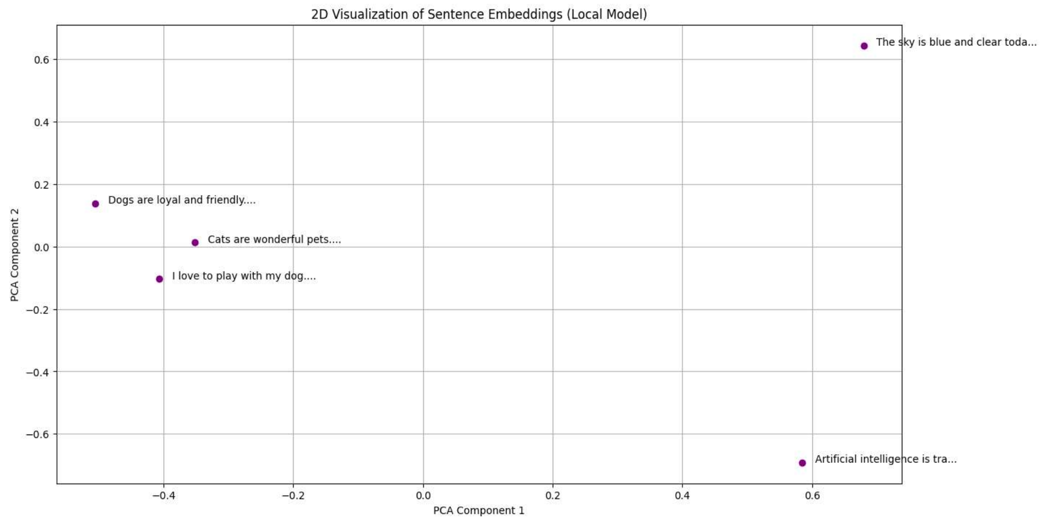
```
Cosine Similarity Between Sentences:

tensor([[ 1.0000,  0.5055,  0.0583,  0.4395,  0.1039],
        [ 0.5055,  1.0000,  0.0176,  0.5127, -0.0222],
        [ 0.0583,  0.0176,  1.0000, -0.0037,  0.0909],
        [ 0.4395,  0.5127, -0.0037,  1.0000,  0.1084],
        [ 0.1039, -0.0222,  0.0909,  0.1084,  1.0000]])
```



```
#Example Sentences ---
sentences = [
    "Cats are wonderful pets.",
    "Dogs are loyal and friendly.",
    "The sky is blue and clear today.",
    "I love to play with my dog.",
    "Artificial intelligence is transforming healthcare."
]
```



```
# Visualize in 2D Using PCA ---
pca = PCA(n_components=2)
reduced_embeddings = pca.fit_transform(sentence_embeddings)

plt.figure(figsize=(15,8))
plt.scatter(reduced_embeddings[:,0], reduced_embeddings[:,1], color='purple')

for i, sentence in enumerate(sentences):
    plt.annotate(sentence[:30] + "...", (reduced_embeddings[i,0]+0.02, reduced_embeddings[i,1]))

plt.title("2D Visualization of Sentence Embeddings (Local Model)")
plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")
plt.grid(True)
plt.show()
```

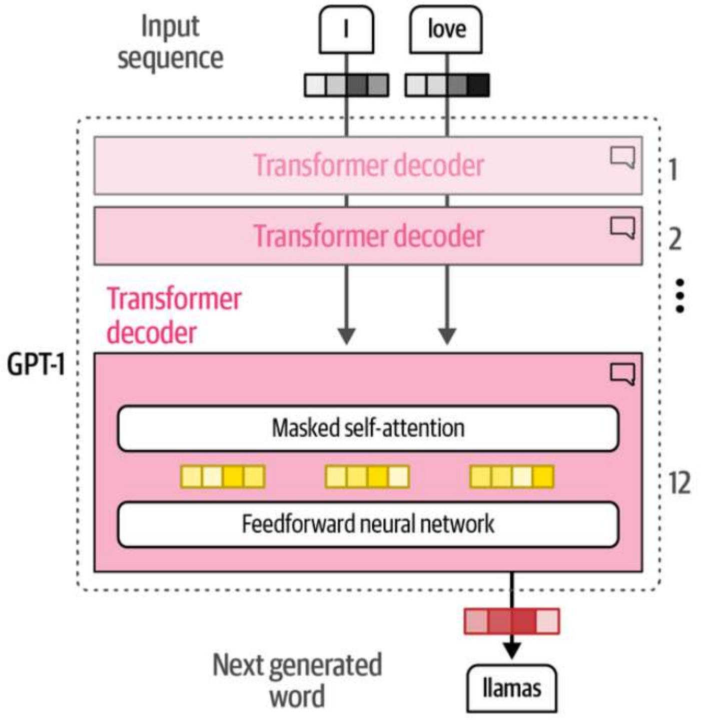


Encoder vs Generative Models

- encoder-only models are called representation models which focus on **representing language**, for instance, by creating embeddings
- Generative models focus primarily on generating text and typically are not trained to generate embeddings

Generative Models: Decoder-Only Models

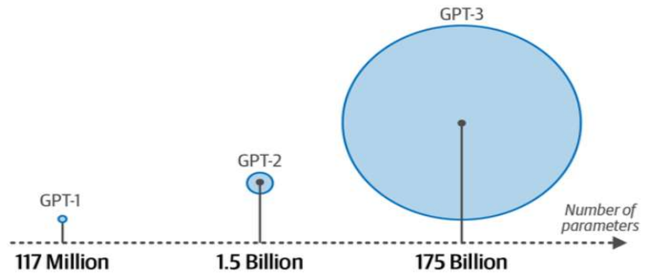
- a decoder-only architecture was proposed in 2018 to **target generative tasks**.
- This architecture was called a Generative Pre-trained Transformer (GPT) for its generative capabilities (it's now known as GPT-1 to distinguish it from later versions).
- it stacks decoder blocks similar to the encoder-stacked architecture of BERT.

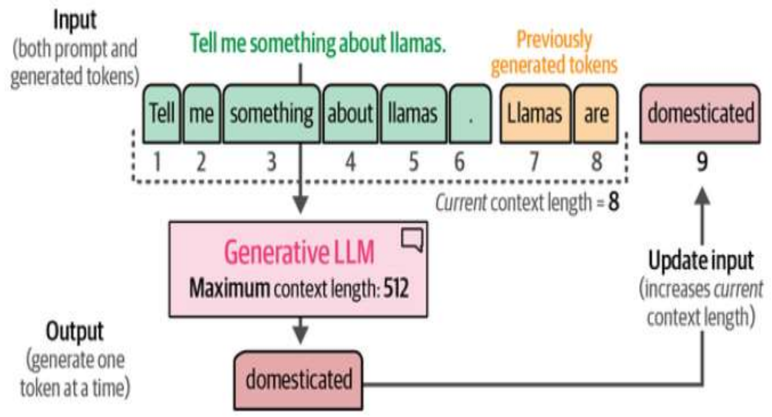


GPTs

- These generative decoder-only models, especially the "larger" models, are commonly referred to as large language models (LLMs).
- Generative LLMs, as sequence-to-sequence machines, take in some text and attempt to autocomplete it.
- the resulting model could take in a user query (prompt) and output a response that would most likely follow that prompt
- A vital part of these completion models **context length or context window.**
- The context length represents the maximum number of tokens the model can process,
- A large context window allows entire documents to be passed to the LLM.
- Note that due to the autoregressive nature of these models, the current context length will increase as new tokens are generated.

GPT models quickly grew in size with each iteration.

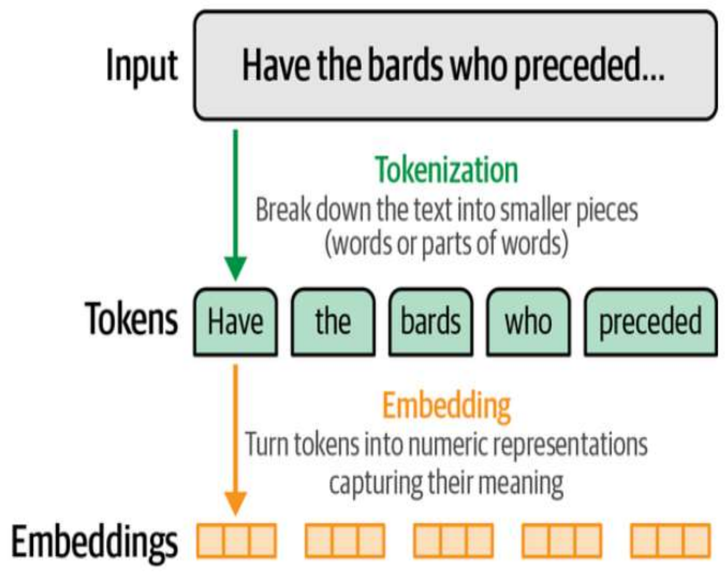



Input
(both prompt and generated tokens)
Tell me something about llamas.
Previously generated tokens
Tell
me
something
about
llamas
.
Llamas
are
domesticated
1
2
3
4
5
6
7
8
9
Current context length = 8
Generative LLM
Maximum context length: 512
Update input
(increases current context length)
Output
(generate one token at a time)
domesticated

How LLMs Work

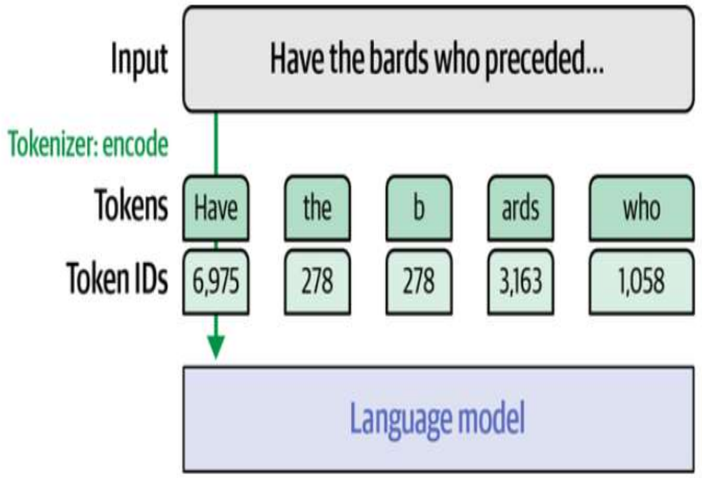
- Tokens and embeddings are two of the central concepts of using LLMs.

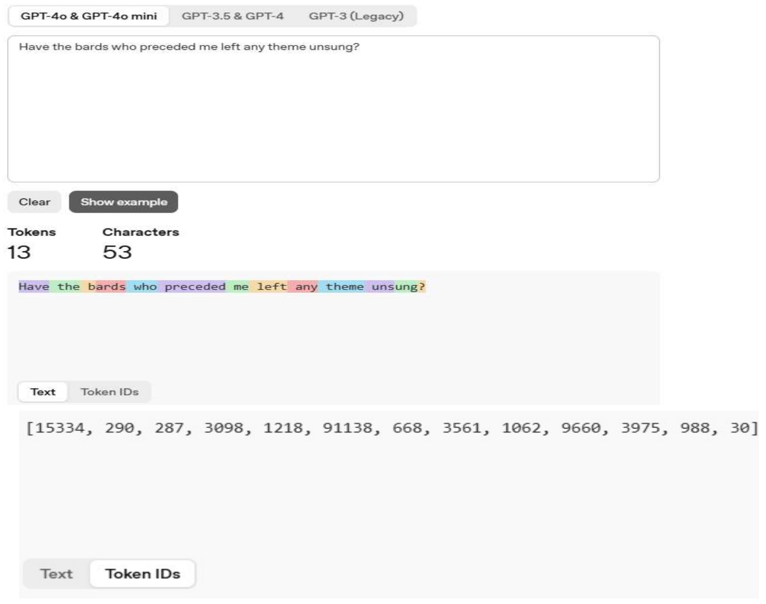
  - LLMs deal with text in small chunks called tokens.
  - For the language model to compute language, it needs to turn tokens into numeric representations called embeddings


Input
Have the bards who preceded...
Tokenization
Break down 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
Embeddings

LLM Tokenization

- A text prompt sent to the model is first broken down into tokens





Have the bards who preceded me left any theme unsung?

Tokens 13
Characters 53

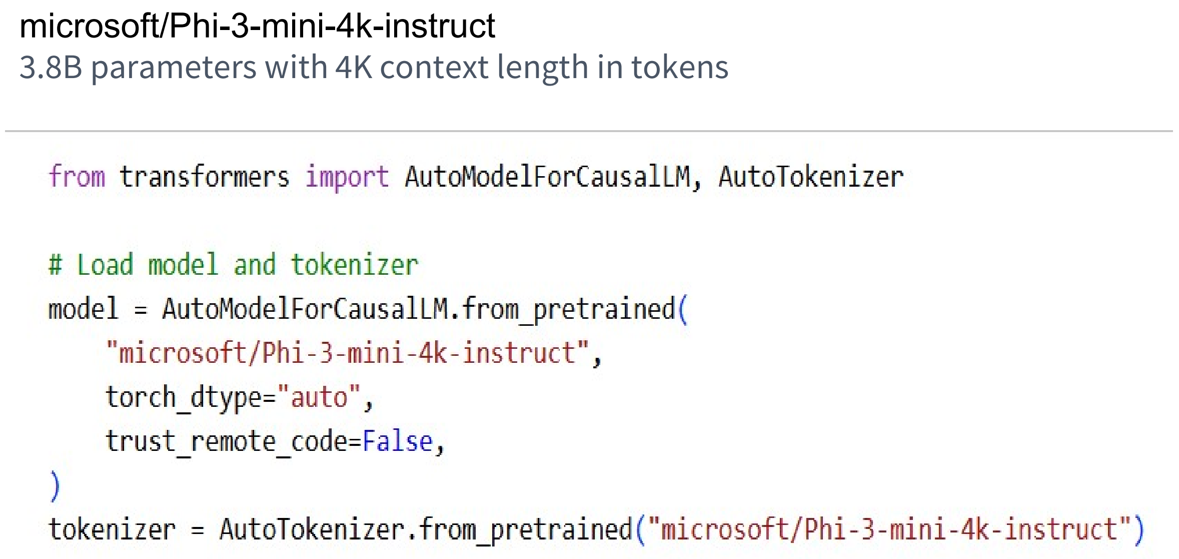
Have the bards who preceded me left any theme unsung?

```
[15334, 290, 287, 3098, 1218, 91138, 668, 3561, 1062, 9660, 3975, 988, 30]
```

You can find an example showing the tokenizer of GPT-4 on the OpenAI Platform.

Demo

- Load generative model itself and underlying tokenizer



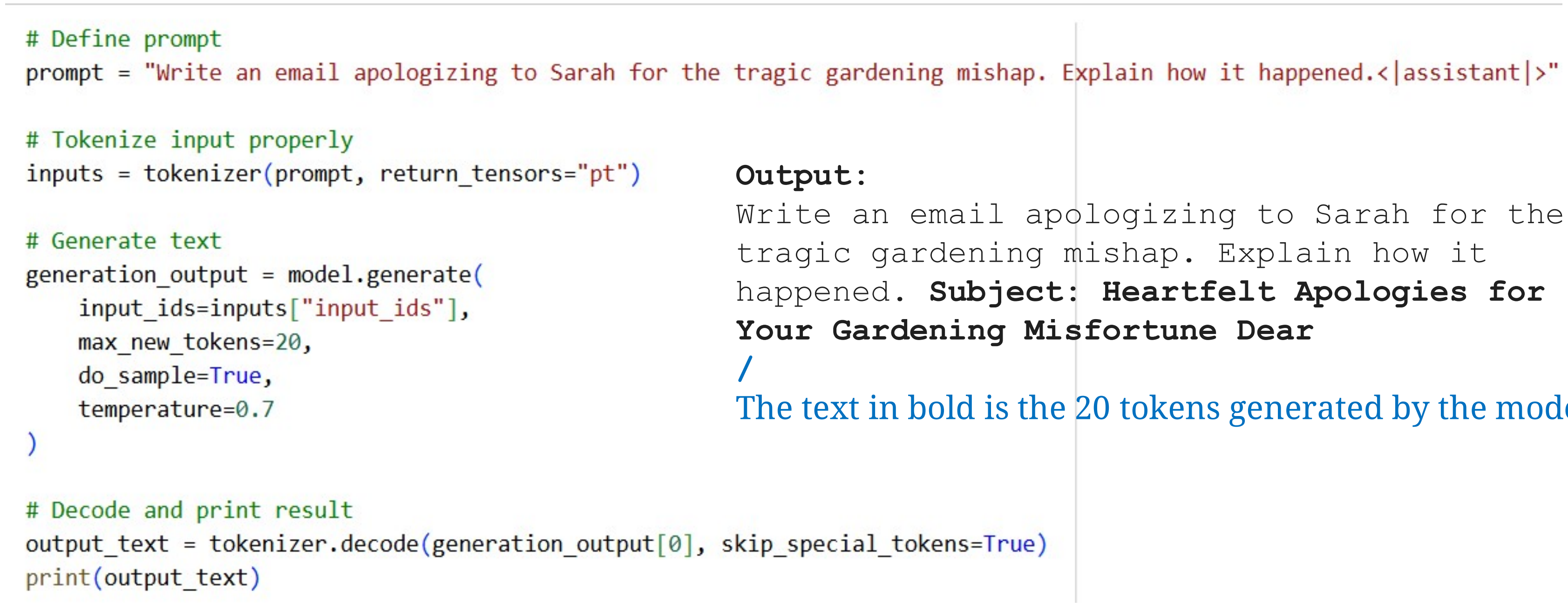
microsoft/Phi-3-mini-4k-instruct
3.8B parameters with 4K context length in tokens

```python
from transformers import AutoModelForCausalLM, AutoTokenizer

# Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Phi-3-mini-4k-instruct",
    torch_dtype="auto",
    trust_remote_code=False,
)
tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-4k-instruct")
```

https://huggingface.co/models?pipeline_tag=text-generation&sort=trending&search=phi-3-4k

Demo

- Define prompt for text generation
- Create Text generation pipeline



```python
# Define prompt
prompt = "Write an email apologizing to Sarah for the tragic gardening mishap. Explain how it happened.<|assistant|>"

# Tokenize input properly
inputs = tokenizer(prompt, return_tensors="pt")

# Generate text
generation_output = model.generate(
    input_ids=inputs["input_ids"],
    max_new_tokens=20,
    do_sample=True,
    temperature=0.7
)

# Decode and print result
output_text = tokenizer.decode(generation_output[0], skip_special_tokens=True)
print(output_text)
```

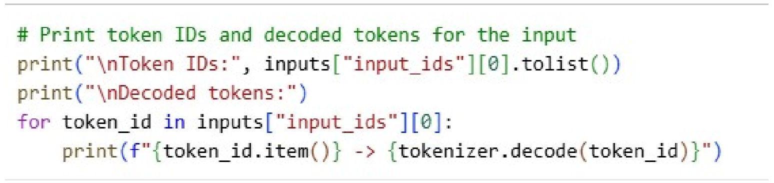
Output:

```
Write an email apologizing to Sarah for the
tragic gardening mishap. Explain how it
happened. Subject: Heartfelt Apologies for
Your Gardening Misfortune Dear
/
```

The text in bold is the 20 tokens generated by the model.

Demo

- Print Token IDs and tokens for input prompt



```python
# Print token IDs and decoded tokens for the input
print("\nToken IDs:", inputs["input_ids"][0].tolist())
print("\nDecoded tokens:")
for token_id in inputs["input_ids"][0]:
    print(f"{token_id.item()} -> {tokenizer.decode(token_id)}")
```

- Output

```
Token IDs: [14350, 385, 4876, 27746,
5281, 304, 19235, 363, 278, 25305,
293, 16423, 292, 286, 728, 481,
29889, 12027, 7420, 920, 372, 9559,
29889, 32001]
```

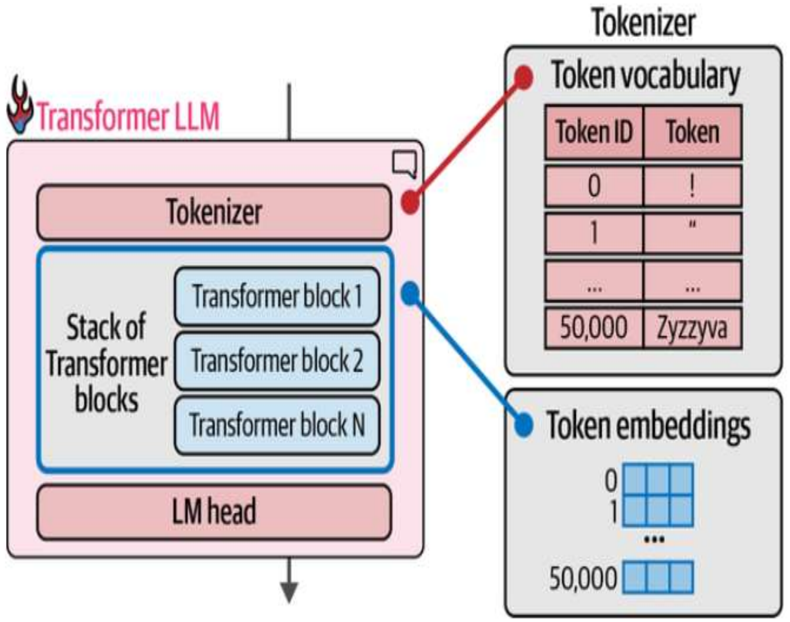
```
Decoded tokens: 14350 -> Write 385 -> an 4876 -> email
27746 -> apolog 5281 -> izing 304 -> to 19235 -> Sarah
363 -> for 278 -> the 25305 -> trag 293 -> ic 16423 ->
garden 292 -> ing 286 -> m 728 -> ish 481 -> ap 29889
-> . 12027 -> Exp 7420 -> lain 920 -> how 372 -> it
9559 -> happened 29889 -> . 32001 -> <|assistant|>
```

Observe how Tokenizer Break Down Text

Note: See the complete code under lab tutorial

LLM Embeddings

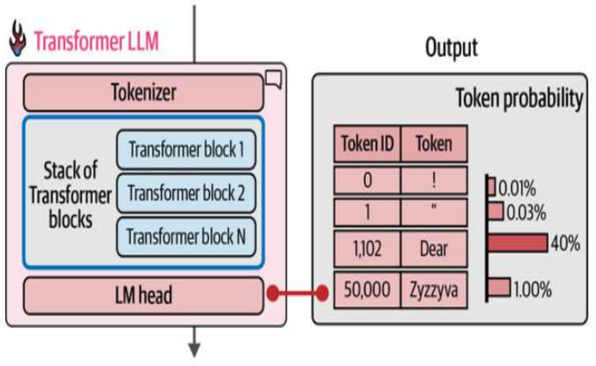
- The tokenizer is followed by the neural network: **a stack of Transformer blocks that do all of the processing.**
- That stack is then followed by the **LM head**, which translates the output of the stack into probability scores for what the **most likely next token is.**
- tokenizer contains a table of tokens— the **tokenizer's vocabulary**.
- The language model holds an embedding vector for each token in the tokenizer's vocabulary
- When we download a pretrained language model, a portion of the model is this embeddings matrix holding all of these vectors.
- Before the beginning of the training process, these vectors are randomly initialized, but the training process assigns them the values that enable the useful behavior they're trained to perform.

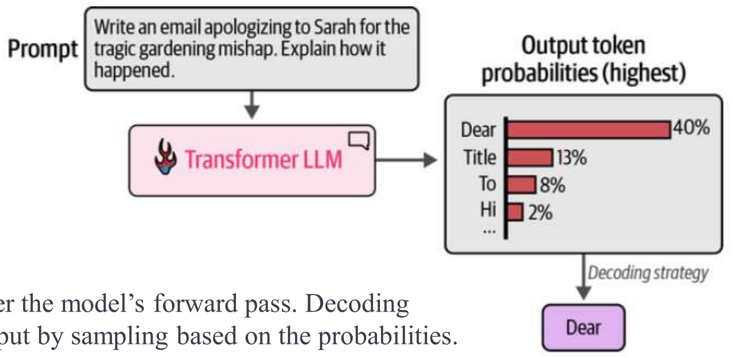


The Forward Pass

- For each generated token, the process flows once through each of the Transformer blocks in the stack in order, then to the LM head, which finally outputs the probability distribution for the next token

- The LM head is a simple neural network layer itself.
- It is one of multiple possible "heads" to attach to a stack of Transformer blocks to build different kinds of systems.
- Other kinds of Transformer heads include sequence classification heads and token classification heads.

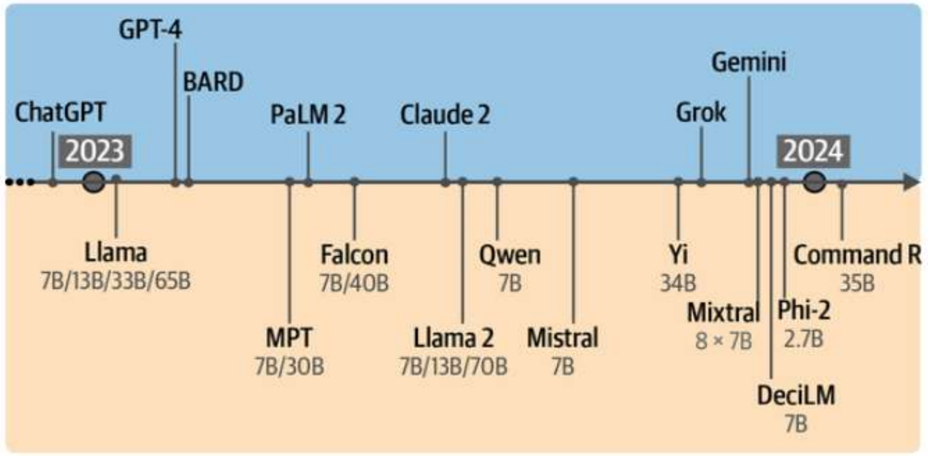



Prompt
Write an email apologizing to Sarah for the tragic gardening mishap. Explain how it happened.
Transformer LLM
Output token probabilities (highest)
Dear
Title
To
Hi
...
40%
13%
8%
2%
Decoding strategy
Dear

The tokens with the highest probability after the model's forward pass. Decoding strategy decides which of the tokens to output by sampling based on the probabilities.

The Year of Generative AI

- LLMs had a tremendous impact on the field and led some to call 2023 The Year of Generative AI with the release, adoption, and media coverage of ChatGPT (GPT-3.5).
- both open source and proprietary LLMs have made their way to the people at an incredible pace.
- These open source base models are often referred to as foundation models and can be fine-tuned for specific tasks, like following instructions.



See Text generation models:
https://huggingface.co/models?pipeline_tag=text-generation&sort=trending

Summry

- Several approaches to transform the text into numerical vectors that are increasingly sophisticated starting from:
  - one-hot encoding, bag of words (BoW), term frequency-inverse document frequency (TF-IDF))
  - until we create vectors of real numbers that represent the meaning of a word (or document) and allow us to conduct operations (word2vec).

- These are the bases that will help us understand how a large language model (LLM) (such as how ChatGPT) works internally.

- Learn text embedding and text generation models from Huggingface:

https://huggingface.co/models?pipeline_tag=text-generation&sort=trending

Next

- Developing LLMs Powered Applications