✨ Member-only story

LLM Architectures Explained: NLP Fundamentals (Part 1)

Vipra Singh · Follow

39 min read · Aug 15, 2024

Listen Share More

Deep Dive into the architecture & building of real-world applications leveraging NLP Models starting from RNN to the Transformers.

Posts in this Series

1. *NLP Fundamentals (This Post)*
2. *Word Embeddings*
3. *RNNs, LSTMs & GRUs*
4. *Sequence to Sequence Models*
5. *Attention Mechanism*
6. *Transformers*
7. *BERT*
8. *GPT*
9. *LLama*
10. *Mistral*

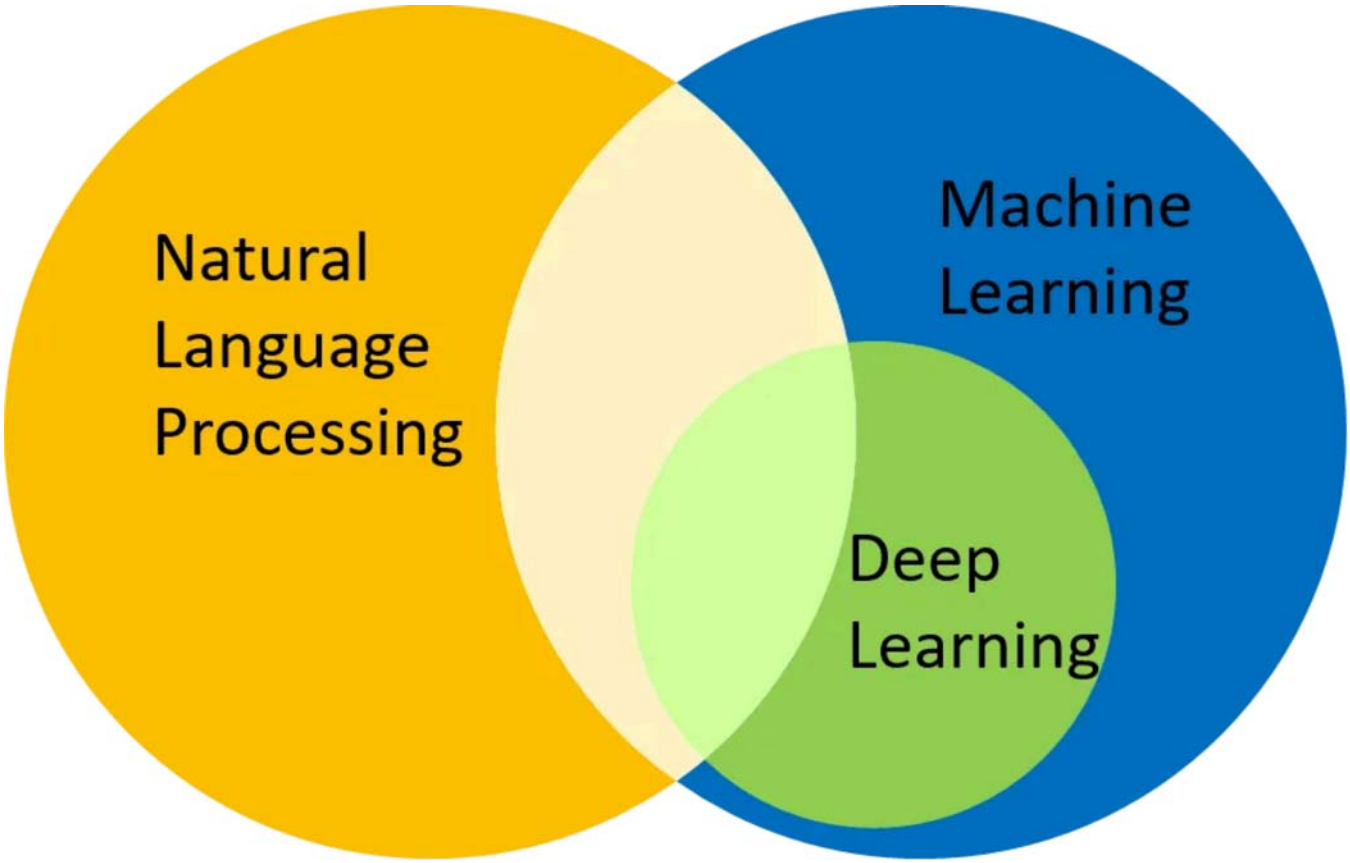
Table of Contents

Welcome to this exciting new blog series!

This blog aims to provide a foundational understanding of Natural Language Processing before delving into them in greater detail. Additionally, the blog offers insights into the evolution of NLP over the years.

1. What is Natural Language Processing (NLP)

Natural language processing (NLP) is the discipline of building machines that can manipulate human language — or data that resembles human language — in the way that it is written, spoken, and organized.

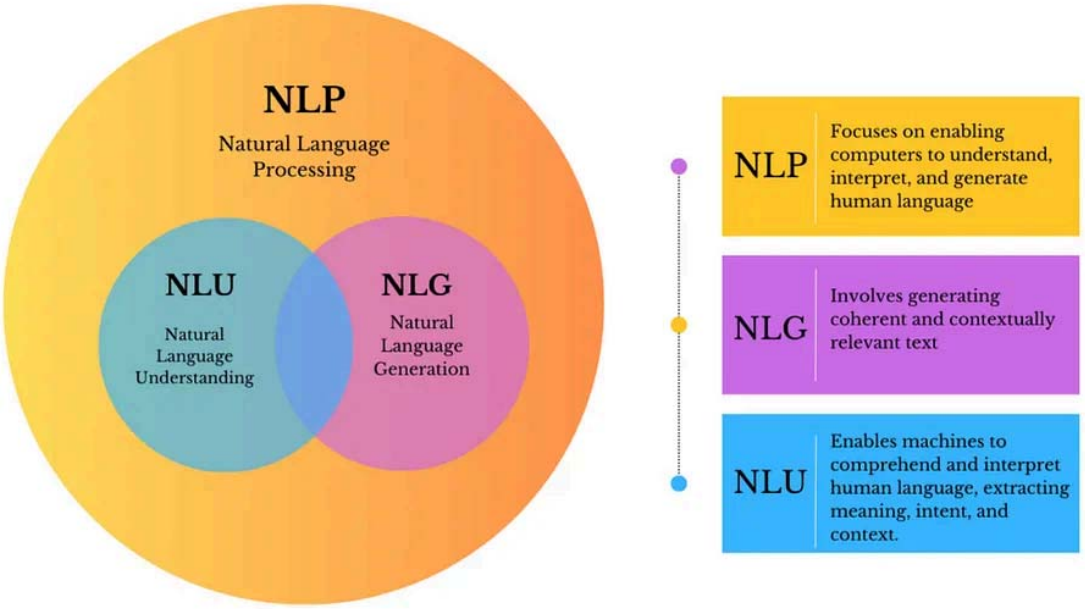


Credits: xoriant

NLP is broadly divided into two overlapping areas:

Natural Language Understanding (NLU), which deals with interpreting the meaning behind text, and

Natural Language Generation (NLG), which focuses on producing text that mimics human writing. Although distinct from speech recognition — which converts spoken language to text — NLP often works in tandem with it.



Credits: GEEKFLARE

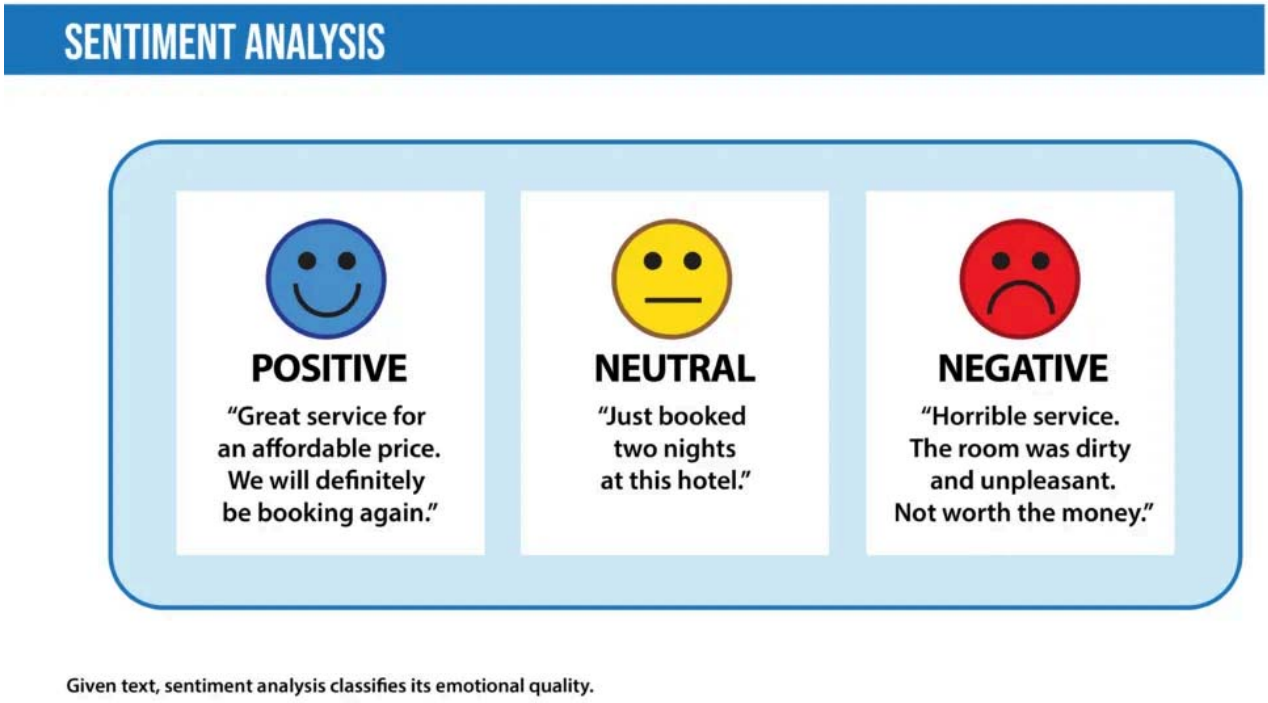
2. Applications of NLP

NLP is used for various language-related tasks, including answering questions, classifying text in various ways, and conversing with users.

Here are some of the tasks that NLP can solve:

Credits: Datasciencedojo

2.1 Sentiment Analysis: This involves determining the emotional tone of text. The input is typically a piece of text, and the output is a probability distribution indicating whether the sentiment is positive, negative, or neutral. Techniques range from traditional methods like TF-IDF and n-grams to deep learning models like BERT and LSTM.



Credits: DeepLearning.AI

2.2 Toxicity Classification: A specialized form of sentiment analysis, toxicity classification identifies hostile intent and categorizes it into specific types like threats or insults. This is used to moderate online content, ensuring safer digital spaces.

2.3 Machine Translation: This automates the translation of text from one language to another. Advanced models like OpenAI's GPT-4 and Google's Transformer-based models are leading the way in making translations more accurate and contextually aware.

2.4 Named Entity Recognition (NER): NER models extract and classify entities like names, organizations, and locations from text. These models are essential in summarizing news and combating disinformation.

spaCy named entity recognition tagging of the first paragraph of Andrew Ng's Wikipedia page. "NORP" stands for nationalities or religious or political groups. Note that spaCy incorrectly labels "AI" as "GPE," for geopolitical entity.

Credits: DeepLearning.AI

2.5 Spam Detection: Spam detection models classify emails as spam or not, helping email services like Gmail filter out unwanted messages. These models often rely on techniques such as logistic regression, naive Bayes, or deep learning.

2.6 Grammatical Error Correction: Models that correct grammatical errors are widely used in tools like Grammarly. They treat grammar correction as a sequence-to-sequence problem, where the input is an incorrect sentence, and the output is a corrected version.

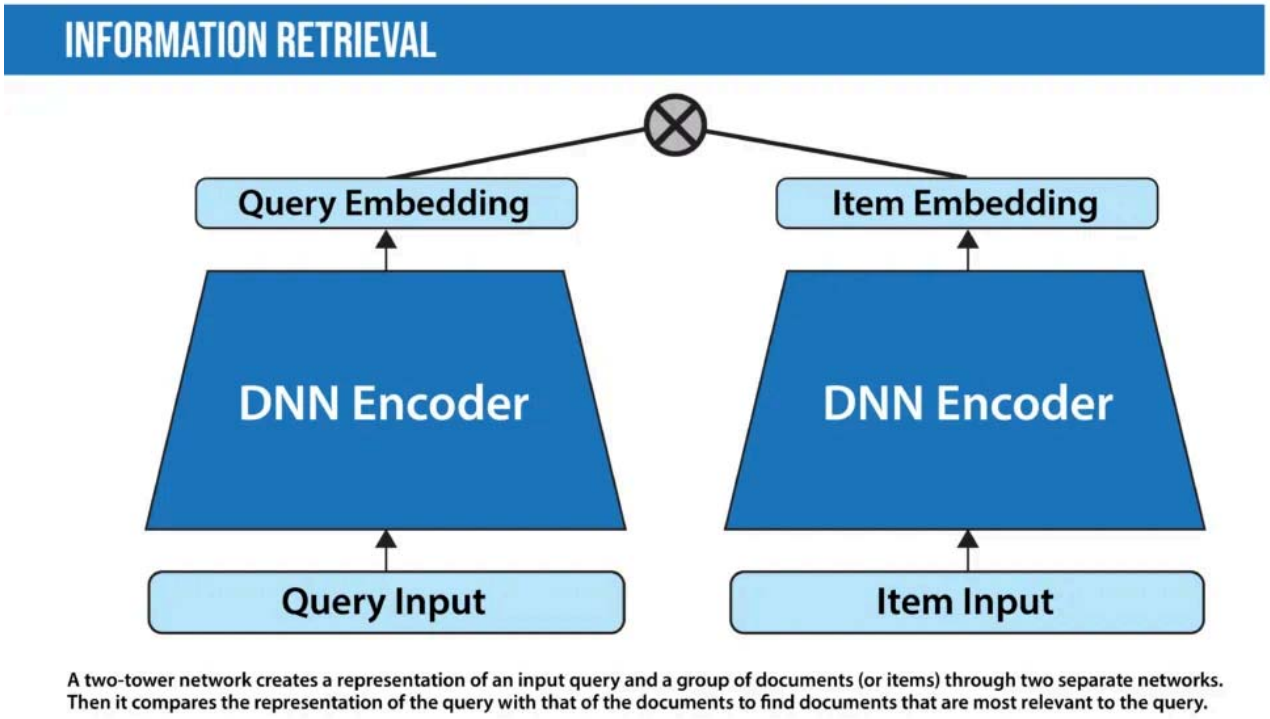
2.7 Topic Modeling: Topic modeling identifies abstract topics within a document corpus. Techniques like Latent Dirichlet Allocation (LDA) are commonly used in legal document analysis and content recommendation systems.

2.8 Text Generation (NLG): NLG models generate human-like text, useful for applications ranging from chatbots to content creation. Modern models like GPT-4 are capable of generating coherent and contextually appropriate text in various genres.

- **Autocomplete:** Autocomplete systems predict the next word in a sequence, used in applications like search engines and messaging apps. Models like GPT-3 have been particularly effective in enhancing autocomplete functionality.

- **Chatbots:** Chatbots simulate human conversations, either through querying databases or generating dialogue. Google's LaMDA is an example of a chatbot capable of engaging in nuanced and human-like conversations.

2.9 Information Retrieval: This involves finding documents relevant to a query, crucial for search engines and recommendation systems. Google's latest models use multimodal approaches to handle text, image, and video data simultaneously.



Credits: DeepLearning.AI

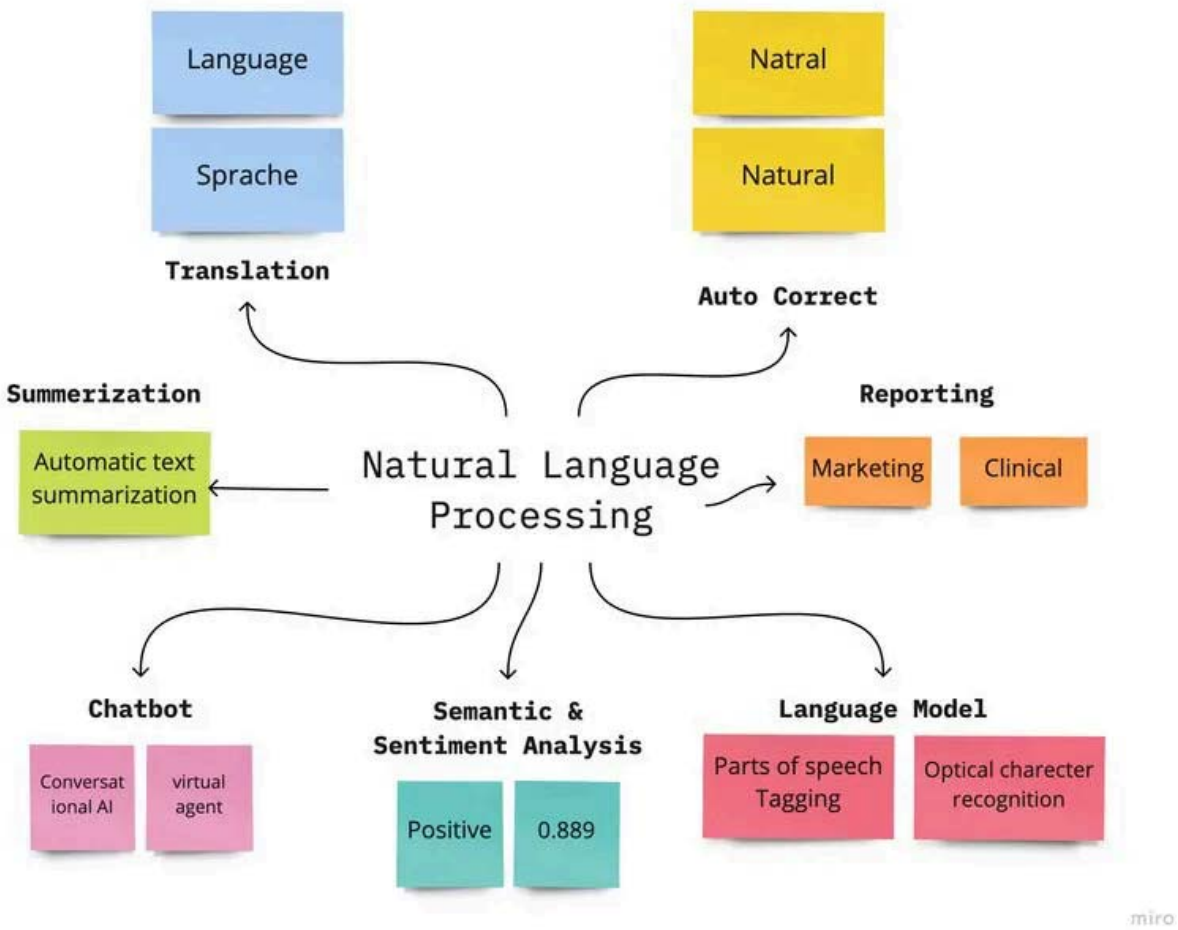
2.10 Summarization is the task of shortening text to highlight the most relevant information. Summarization is divided into two method classes:

- **Extractive summarization** focuses on extracting the most important sentences from a long text and combining these to form a summary. Typically, extractive summarization scores each sentence in an input text and then selects several sentences to form the summary.
- **Abstractive summarization** produces a summary by paraphrasing. This is similar to writing the abstract which includes words and sentences that are not present in the original text. Abstractive summarization is usually modeled as a sequence-to-sequence task, where the input is a long-form text and the output is a summary.

2.11 Question Answering (QA) deals with answering questions posed by humans in a natural language. One of the most notable examples of question answering was Watson, which 2011 played the television game-show *Jeopardy* against human champions and won by substantial margins. Generally, question-answering tasks come in two flavors:

- **Multiple choice:** The multiple-choice question problem is composed of a question and a set of possible answers. The learning task is to pick the correct answer.
- **Open domain:** In open-domain question answering, the model provides answers to questions in natural language without any options provided, often by querying a large number of texts.

3. NLP Terms



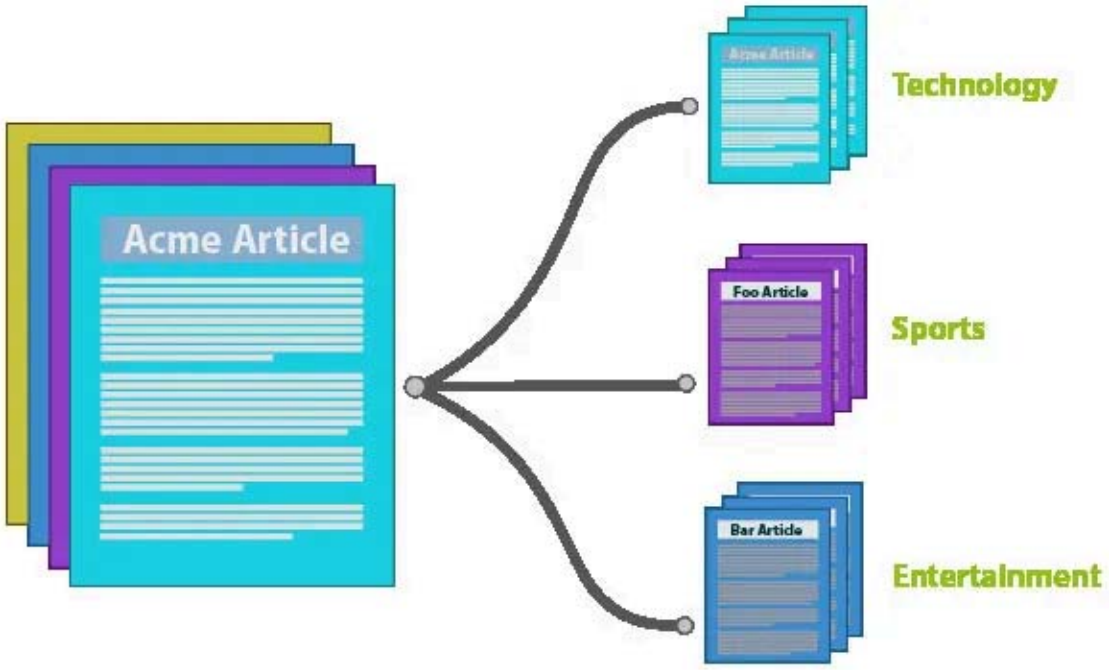
Credits: Md Mahmud Hasan

3.1 Document

A **document** is a single piece of text, which can be anything from a single sentence to an entire book. It is the basic unit of text that NLP models process. Documents can be diverse in nature, such as emails, web pages, articles, or tweets.

Example:

- A single news article from a newspaper.
- A tweet: “Just watched an amazing movie!”
- An email: “Dear John, I hope this email finds you well…”



Credits: Parsa Ghaffari

3.2 Corpus (Corpora)

A **corpus** (plural: corpora) is a large collection of documents. It serves as the dataset on which NLP models are trained and evaluated. A corpus typically contains documents that are related by topic, language, or genre and is used to analyze linguistic patterns and build statistical models.

Example:

- A collection of all articles from a specific newspaper over a year.
- A dataset of customer reviews from an e-commerce website.
- The Gutenberg Corpus: A collection of literary texts from Project Gutenberg.

3.3 Feature

A **feature** is a measurable property or characteristic of the text that is used in machine learning models. Features are extracted from documents and can represent various aspects of the text, such as the presence of specific words, the length of sentences, or the occurrence of particular patterns.

Example:

Bag-of-Words (BoW): Each word in the vocabulary is a feature, and the value is the count of the word in the document.

- Document: “I love NLP.”
- Features: {“I”: 1, “love”: 1, “NLP”: 1}

Term Frequency-Inverse Document Frequency (TF-IDF): A statistical measure used to evaluate the importance of a word in a document relative to a corpus.

- Document: “Machine learning is fun.”
- Features (TF-IDF scores): {“Machine”: 0.5, “learning”: 0.5, “is”: 0.1, “fun”: 0.7}

Part of Speech (POS) Tags: Features indicating the grammatical category of each word (e.g., noun, verb, adjective).

- Document: “The quick brown fox jumps.”
- Features: {“The”: “DET”, “quick”: “ADJ”, “brown”: “ADJ”, “fox”: “NOUN”, “jumps”: “VERB”}

For Example, Let’s consider the 2 documents shown below:

```
Sentences:
Dog hates a cat. It loves to go out and play.
Cat loves to play with a ball.
```

We can build a corpus from the above 2 documents just by combining them.

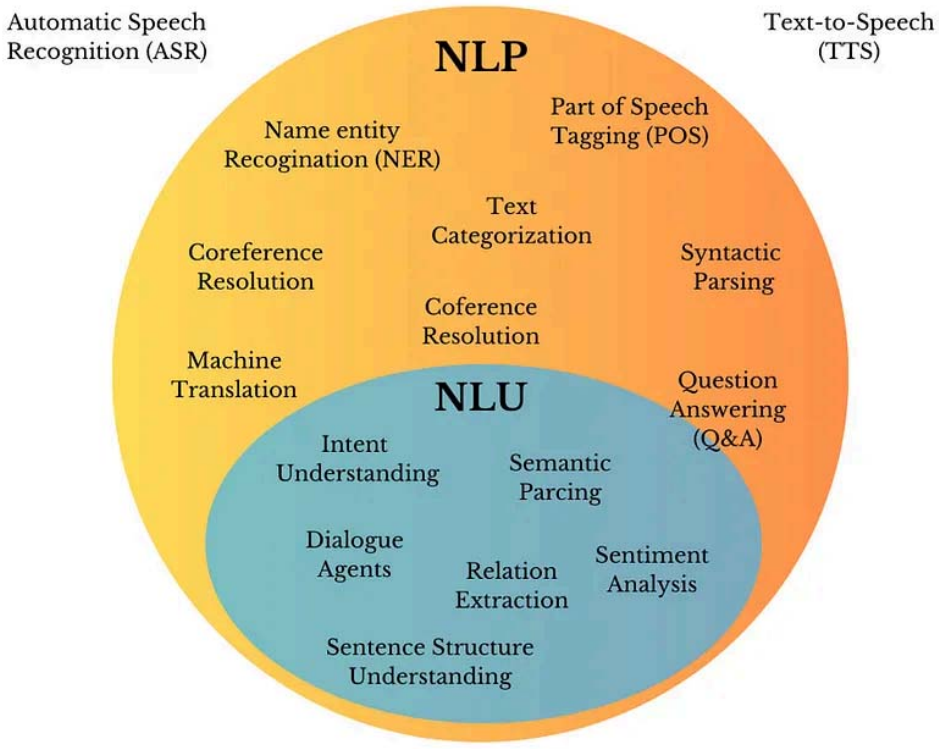
```
Corpus = “Dog hates a cat. It loves to go out and play. Cat loves to play with
```

And features will be all unique words:

```
Fetaures: [‘and’, ‘ball’, ‘cat’, ‘dog’, ‘go’, ‘hates’, ‘it’, ‘loves’, ‘out’, ‘p
```

We will call it a feature vector. Here we remember that we will remove ‘a’ by considering it a single character.

4. How NLP works?



Credits: GEEKFLARE

NLP models work by finding relationships between the constituent parts of language — for example, the letters, words, and sentences found in a text dataset. NLP architectures use various methods for data preprocessing, feature extraction, and modeling.

Let's simplify and break down these steps for better understanding.

4.1 Data Pre-processing

The first step of understanding NLP focuses on the meaning of dialogue and discourse within a contextual framework. The primary goal is to facilitate meaningful conversations between a voicebot and a human.

Ex- Giving commands to chatbots, such as "show me the best recipes" or "play party music," falls within the scope of this step. It involves understanding and responding to user requests within the context of the ongoing conversation.

Before a model processes text for a specific task, the text often needs to be preprocessed to improve model performance or to turn words and characters into a format the model can understand. Various techniques may be used in this data preprocessing:

4.1.1 Tokenization

The process of breaking down text into smaller units called tokens, which can be words, subwords, or characters.

Types:

- **Word Tokenization**: Splitting text into individual words. Example:

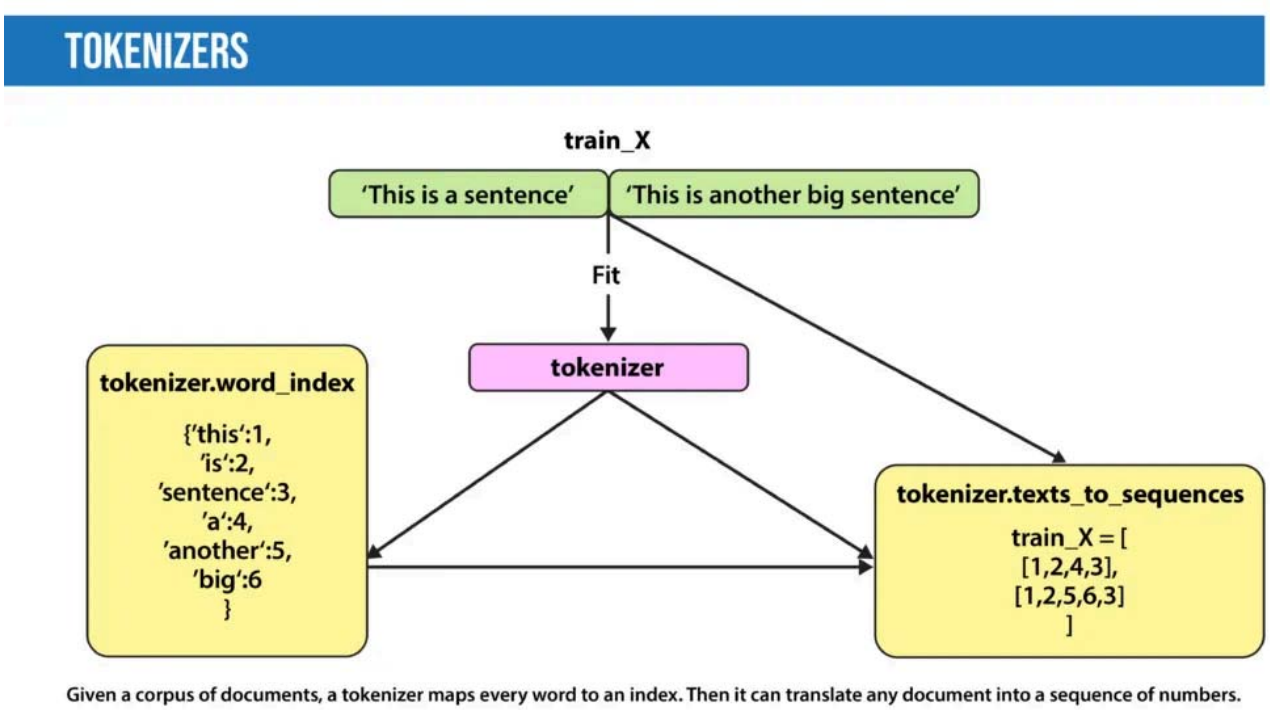
"I study Machine Learning on GeeksforGeeks." will be word-tokenized as
['I', 'study', 'Machine', 'Learning', 'on', 'GeeksforGeeks', '.']

- **Sentence Tokenization**: Splitting text into individual sentences. Example:

"I study Machine Learning on GeeksforGeeks. Currently, I'm studying NLP"
will be sentence-tokenized as
['I study Machine Learning on GeeksforGeeks.', 'Currently, I'm studying NLP.']

- **Subword Tokenization**: Breaking words into smaller units like prefixes, suffixes, or individual characters.

Importance: Tokenization is the first step in many NLP pipelines and affects the subsequent processing stages.



Credits: DeepLearning.AI

4.1.2 Stemming

Definition: The process of reducing words to their root form by stripping suffixes and prefixes.

Example: The words “running” and “runner” are stemmed to “run”.

Difference from Lemmatization: Stemming is a more crude technique compared to lemmatization and may not always produce real words.

4.1.3 Lemmatisation

Definition: The process of reducing a word to its base or dictionary form, called a lemma.

Example: The words “running” and “ran” are lemmatized to “run”.

Importance: Helps in understanding the underlying meaning of words by grouping different forms of a word.

Stemmers are faster and computationally less expensive than lemmatizers.

```
from nltk.stem import WordNetLemmatizer
# create an object of class WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
```

```
print(lemmatizer.lemmatize("plays", 'v'))
print(lemmatizer.lemmatize("played", 'v'))
print(lemmatizer.lemmatize("play", 'v'))
print(lemmatizer.lemmatize("playing", 'v'))
```

Output :

```
play
play
play
play
```

Lemmatization involves grouping together the inflected forms of the same word. This way, we can reach out to the base form of any word that will be meaningful in nature. The base from here is called the Lemma.

```
from nltk.stem import WordNetLemmatizer

# create an object of class WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
print(lemmatizer.lemmatize("Communication", 'v'))
```

Output :

```
Communication
```

Lemmatizers are slower and computationally more expensive than stemmers.

4.1.4 Normalization

Normalization in Natural Language Processing (NLP) refers to the process of converting text into a standard form. The purpose of normalization is to ensure consistency in how text data is processed, allowing NLP models to better understand and interpret the text. Some common normalization techniques include

converting text to lowercase, stemming, lemmatization, and removing stop words or punctuation.

Lowercasing:

- Converts all characters in the text to lowercase to ensure that words like “Apple” and “apple” are treated as the same word.
- **Example:** `"Apple" -> "apple"`

Removing Punctuation:

- Punctuation marks do not usually carry significant meaning in NLP tasks, so they are often removed.
- **Example:** `"Hello, world!" -> "Hello world"`

Removing Stop Words:

- Stop words are common words (like “and,” “the,” “is”) that may not contribute much to the meaning of the text and are often removed.
- **Example:** `"This is a pen" -> "pen"`

```
import nltk
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer, WordNetLemmatizer
import re

# Download necessary NLTK data
nltk.download('punkt')
nltk.download('wordnet')
nltk.download('stopwords')

# Sample text
text = "Running is better than walking! Apples and oranges are different."
# Lowercasing
text_lower = text.lower()
print("Lowercased text:", text_lower)

# Removing punctuation
text_no_punct = re.sub(r'[^\w\s]', '', text_lower)
print("Text without punctuation:", text_no_punct)

# Tokenization
```

```
words = nltk.word_tokenize(text_no_punct)
print("Tokenized words:", words)

# Removing stop words
stop_words = set(stopwords.words('english'))
words_no_stop = [word for word in words if word not in stop_words]
print("Text without stopwords:", words_no_stop)

# Stemming
ps = PorterStemmer()
words_stemmed = [ps.stem(word) for word in words_no_stop]
print("Stemmed words:", words_stemmed)

# Lemmatization
lemmatizer = WordNetLemmatizer()
words_lemmatized = [lemmatizer.lemmatize(word) for word in words_no_stop]
print("Lemmatized words:", words_lemmatized)
```

Output:

```
Lowercased text: running is better than walking! apples and oranges are differe
Text without punctuation: running is better than walking apples and oranges are
Tokenized words: ['running', 'is', 'better', 'than', 'walking', 'apples', 'and'
Text without stopwords: ['running', 'better', 'walking', 'apples', 'oranges', '
Stemmed words: ['run', 'better', 'walk', 'appl', 'orang', 'differ']
Lemmatized words: ['running', 'better', 'walking', 'apple', 'orange', 'differer
```

4.1.5 Part of Speech (POS) Tagging

Definition: The process of assigning parts of speech to each word in a sentence, such as nouns, verbs, adjectives, etc.

Importance: Helps in understanding the grammatical structure and meaning of sentences.

Example:
"GeeksforGeeks is a Computer Science platform."
Let's see how NLTK's POS tagger will tag this sentence.

```
from nltk import pos_tag
from nltk import word_tokenize
```

```
text = "GeeksforGeeks is a Computer Science platform."
tokenized_text = word_tokenize(text)
tags = tokens_tag = pos_tag(tokenized_text)
tags
```

Output :

```
[('GeeksforGeeks', 'NNP'),
 ('is', 'VBZ'),
 ('a', 'DT'),
 ('Computer', 'NNP'),
 ('Science', 'NNP'),
 ('platform', 'NN'),
 ('.', '.')]
```

4.2 Feature Extraction

Most conventional machine-learning techniques work on the features — generally numbers that describe a document in relation to the corpus that contains it — created by either Bag-of-Words, TF-IDF, or generic feature engineering such as document length, word polarity, and metadata (for instance, if the text has associated tags or scores). More recent techniques include Word2Vec, GLoVE, and learning the features during the training process of a neural network.

4.2.1 Bag-of-Words (BoW)

Definition: A representation of text data where each document is represented as a bag (multiset) of its words, disregarding grammar and word order but keeping multiplicity.

The Bag of Words model is a commonly used technique in NLP where each word in the text is represented as a feature, and its frequency is used as the feature value.

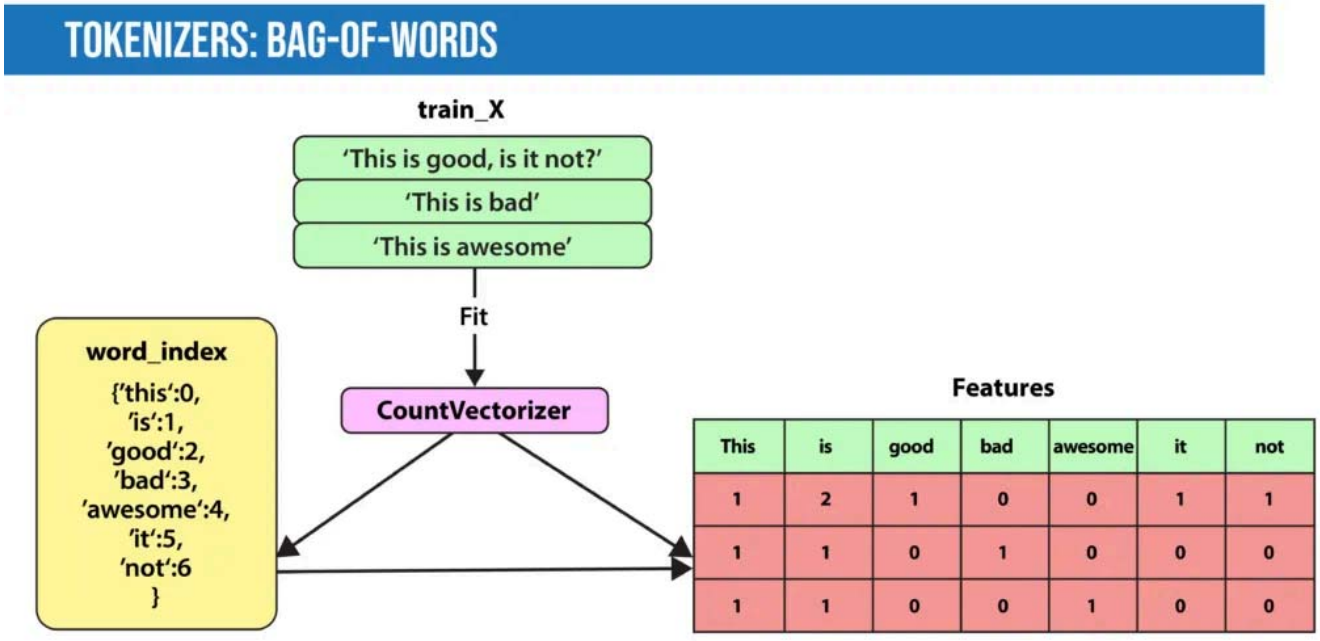
Importance: Simplifies text processing and is used in various NLP applications such as text classification and information retrieval.

```
from sklearn.feature_extraction.text import CountVectorizer

# Sample text data
```

```
documents = [
    "Barack Obama was the 44th President of the United States",
    "The President lives in the White House",
    "The United States has a strong economy"
]

# Initialize the CountVectorizer
vectorizer = CountVectorizer()

# Fit the model and transform the documents into a Bag of Words
bow_matrix = vectorizer.fit_transform(documents)

# Get the feature names (unique words in the corpus)
feature_names = vectorizer.get_feature_names_out()

# Convert the Bag of Words matrix into an array
bow_array = bow_matrix.toarray()

# Display the Bag of Words
print("Feature Names (Words):", feature_names)
print("\nBag of Words Representation:")
print(bow_array)
```

Output:

```
Feature Names (Words): ['44th' 'barack' 'economy' 'has' 'house' 'in' 'lives' 'c
 'president' 'states' 'strong' 'the' 'united' 'was' 'white']

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



Bag-of-Words (through the CountVectorizer method) encodes the total number of times a document uses each word in the associated corpus.

Credits: DeepLearning.AI

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

Definition: TF-IDF (Term Frequency-Inverse Document Frequency) is a numerical statistic used to evaluate the importance of a word in a document relative to a collection of documents (or corpus). The basic idea is that if a word appears frequently in a document but not in many other documents, it should be given more importance.

The TF-IDF score for a term t in a document d within a corpus D is calculated as the product of two metrics: **Term Frequency (TF)** and **Inverse Document Frequency (IDF)**.

1. Term Frequency (TF)

Term Frequency (TF) measures how frequently a term appears in a document. It is often normalized by the total number of terms in the document to prevent bias toward longer documents.

$$\mathrm{TF}(t,d) = \frac{\text{Number of times term t appears in document d}}{\text{Total number of terms in document d}}$$

2. Inverse Document Frequency (IDF)

Inverse Document Frequency (IDF) measures how important a term is in the entire corpus. It decreases the weight of terms that appear in many documents and increases the weight of terms that appear in fewer documents.

$$\text{IDF}(t, D) = \log\left(\frac{\text{Total number of documents in corpus } D}{\text{Number of documents where term } t \text{ appears} + 1}\right)$$

- The "+1" is added to the denominator to prevent division by zero in case the term doesn't appear in any document.

3. Combining TF and IDF: TF-IDF

The TF-IDF score is calculated by multiplying the **TF** value with the **IDF** value for a term *t* in a document *d*:

$$\text{TF-IDF}(t, d, D) = \text{TF}(t, d) \times \text{IDF}(t, D)$$

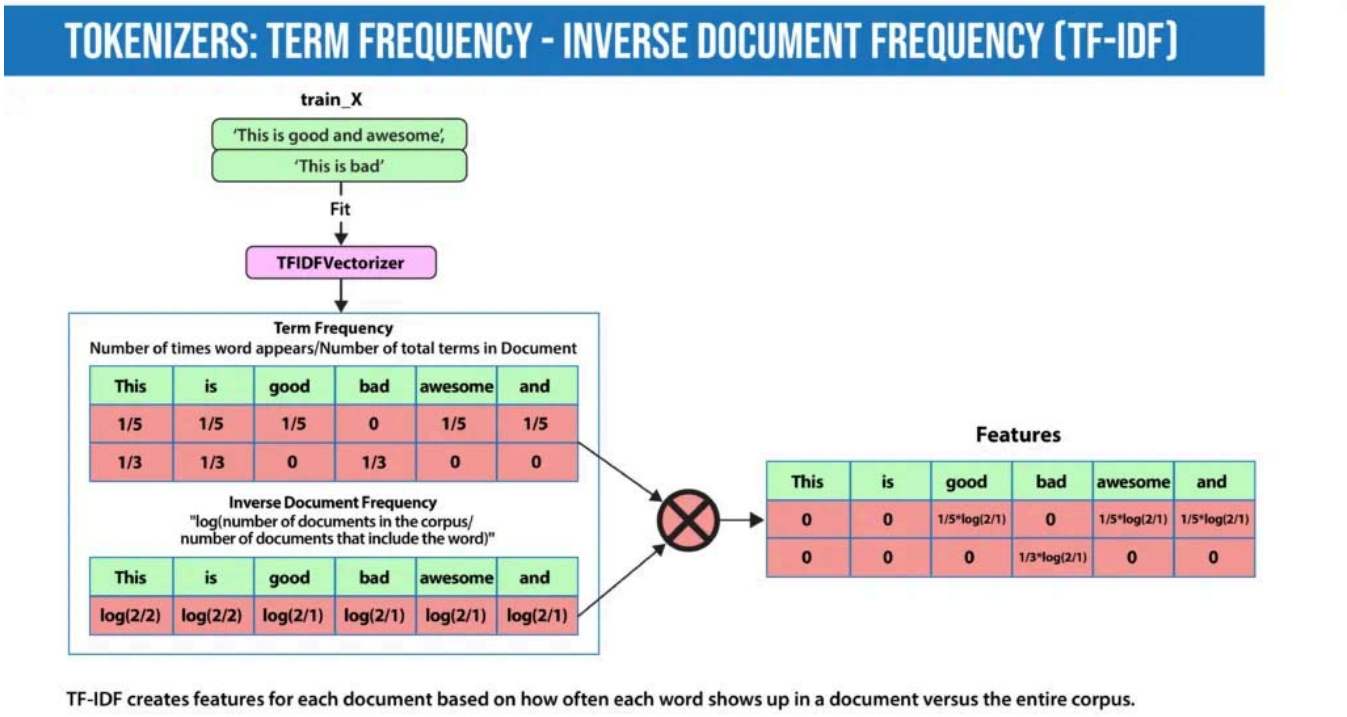
Importance: Helps in identifying important words in documents and is commonly used in information retrieval and text mining.

```
from sklearn.feature_extraction.text import TfidfVectorizer

# Sample text data (documents)
documents = [
    "The cat sat on the mat.",
    "The cat sat on the bed.",
    "The dog barked."
]

# Initialize the TfidfVectorizer
vectorizer = TfidfVectorizer()

# Fit the model and transform the documents into TF-IDF representation
tfidf_matrix = vectorizer.fit_transform(documents)

# Get the feature names (unique words in the corpus)
feature_names = vectorizer.get_feature_names_out()

# Convert the TF-IDF matrix into an array
tfidf_array = tfidf_matrix.toarray()

# Display the TF-IDF matrix
print("Feature Names (Words):", feature_names)
```

```
print("\nTF-IDF Matrix:")
print(tfidf_array)
```

Output:

```
Feature Names (Words): ['barked' 'bed' 'cat' 'dog' 'mat' 'on' 'sat' 'the']

TF-IDF Matrix:
[[0.         0.         0.37420726 0.         0.49203758 0.37420726
  0.37420726 0.58121064]
 [0.         0.49203758 0.37420726 0.         0.         0.37420726
  0.37420726 0.58121064]
 [0.65249088 0.         0.         0.65249088 0.         0.
  0.         0.38537163]]
```



TF-IDF creates features for each document based on how often each word shows up in a document versus the entire corpus.

Credits: DeepLearning.AI

4.2.3 N-grams

Definition: Contiguous sequences of n items (typically words or characters) from a given text or speech.

Types:

- **Unigram:** Single word.

- **Bigram:** Pair of words.
- **Trigram:** Sequence of three words.

Importance: Captures context and word dependencies in text.

```
import nltk
from nltk.util import ngrams
from collections import Counter

# Sample text data
text = "The quick brown fox jumps over the lazy dog"

# Tokenize the text into words
tokens = nltk.word_tokenize(text)

# Generate Unigrams (1-gram)
unigrams = list(ngrams(tokens, 1))
print("Unigrams:")
print(unigrams)

# Generate Bigrams (2-gram)
bigrams = list(ngrams(tokens, 2))
print("\nBigrams:")
print(bigrams)

# Generate Trigrams (3-gram)
trigrams = list(ngrams(tokens, 3))
print("\nTrigrams:")
print(trigrams)

# Count frequency of each n-gram (for demonstration)
unigram_freq = Counter(unigrams)
bigram_freq = Counter(bigrams)
trigram_freq = Counter(trigrams)
# Print frequencies (optional)
print("\nUnigram Frequencies:")
print(unigram_freq)
print("\nBigram Frequencies:")
print(bigram_freq)
print("\nTrigram Frequencies:")
print(trigram_freq)
```

Output:

```
Unigrams:
[('The',), ('quick',), ('brown',), ('fox',), ('jumps',), ('over',), ('the',), (

Bigrams:
[('The', 'quick'), ('quick', 'brown'), ('brown', 'fox'), ('fox', 'jumps'), ('ju

Trigrams:
[('The', 'quick', 'brown'), ('quick', 'brown', 'fox'), ('brown', 'fox', 'jumps'

Unigram Frequencies:
Counter({('The',): 1, ('quick',): 1, ('brown',): 1, ('fox',): 1, ('jumps',): 1,

Bigram Frequencies:
Counter({('The', 'quick'): 1, ('quick', 'brown'): 1, ('brown', 'fox'): 1, ('fox

Trigram Frequencies:
Counter({('The', 'quick', 'brown'): 1, ('quick', 'brown', 'fox'): 1, ('brown',
```

4.2.4 Word Embeddings

Definition: Word embeddings are dense vector representations of words that capture their meanings, syntactic properties, and relationships with other words. Unlike traditional methods like Bag of Words or TF-IDF, which treat words as discrete entities, word embeddings map words into a continuous vector space where semantically similar words are located near each other.

Examples: Word2Vec, GloVe, FastText.

Importance: Captures semantic relationships between words and is used in various NLP models and applications.

```
import gensim
from gensim.models import Word2Vec
from nltk.tokenize import word_tokenize
import nltk

# Sample text data
text = [
    "The cat sat on the mat",
    "The dog barked at the cat",
    "The cat chased the mouse",
    "The dog chased the cat",
]
```

```
# Tokenize the sentences into words
tokenized_text = [word_tokenize(sentence.lower()) for sentence in text]

# Train a Word2Vec model on the tokenized text
model = Word2Vec(tokenized_text, vector_size=100, window=5, min_count=1, sg=0)

# Get the word embeddings for a specific word
cat_vector = model.wv['cat']

# Print the word embedding for 'cat'
print("Word Embedding for 'cat':")
print(cat_vector)

# Find words most similar to 'cat'
similar_words = model.wv.most_similar('cat', topn=3)
print("\nWords most similar to 'cat':")
print(similar_words)
```

Output:

```
Word Embedding for 'cat':
[-8.6196875e-03  3.6657380e-03  5.1898835e-03  5.7419385e-03
  7.4669183e-03 -6.1676754e-03  1.1056137e-03  6.0472824e-03
 -2.8400505e-03 -6.1735227e-03 -4.1022300e-04 -8.3689485e-03
 -5.6000124e-03  7.1045388e-03  3.3525396e-03  7.2256695e-03
  6.8002474e-03  7.5307419e-03 -3.7891543e-03 -5.6180597e-04
  2.3483764e-03 -4.5190323e-03  8.3887316e-03 -9.8581640e-03
  6.7646410e-03  2.9144168e-03 -4.9328315e-03  4.3981876e-03
 -1.7395747e-03  6.7113843e-03  9.9648498e-03 -4.3624435e-03
 -5.9933780e-04 -5.6956373e-03  3.8508223e-03  2.7866268e-03
  6.8910765e-03  6.1010956e-03  9.5384968e-03  9.2734173e-03
  7.8980681e-03 -6.9895042e-03 -9.1558648e-03 -3.5575271e-04
 -3.0998408e-03  7.8943167e-03  5.9385742e-03 -1.5456629e-03
  1.5109634e-03  1.7900408e-03  7.8175711e-03 -9.5101865e-03
 -2.0553112e-04  3.4691966e-03 -9.3897223e-04  8.3817719e-03
  9.0107834e-03  6.5365066e-03 -7.1162102e-04  7.7104042e-03
 -8.5343346e-03  3.2071066e-03 -4.6379971e-03 -5.0889552e-03
  3.5896183e-03  5.3703394e-03  7.7695143e-03 -5.7665063e-03
  7.4333609e-03  6.6254963e-03 -3.7098003e-03 -8.7456414e-03
  5.4374672e-03  6.5097557e-03 -7.8755023e-04 -6.7098560e-03
 -7.0859254e-03 -2.4970602e-03  5.1432536e-03 -3.6652375e-03
 -9.3700597e-03  3.8267397e-03  4.8844791e-03 -6.4285635e-03
  1.2085581e-03 -2.0748770e-03  2.4403334e-05 -9.8835090e-03
  2.6920044e-03 -4.7501065e-03  1.0876465e-03 -1.5762246e-03
  2.1966731e-03 -7.8815762e-03 -2.7171839e-03  2.6631986e-03
  5.3466819e-03 -2.3915148e-03 -9.5100943e-03  4.5058788e-03]
```

```
Words most similar to 'cat':
[('dog', 0.06797593832015991), ('on', 0.033640578389167786), ('at', 0.009391162
```

4.2.5 Contextual Word Embeddings

Definition: Contextual word embeddings capture the meaning of a word based on its context in a sentence. Unlike traditional word embeddings (like Word2Vec), which provide a single vector for each word regardless of context, contextual embeddings produce different vectors for a word depending on the surrounding words. These embeddings are generated using models like BERT (Bidirectional Encoder Representations from Transformers), which take into account the entire sentence when computing word representations.

Examples: ELMo, BERT, GPT.

Importance: Provides more accurate word representations by considering the surrounding context.

```
from transformers import BertModel, BertTokenizer
import torch

# Load pre-trained BERT model and tokenizer
model_name = 'bert-base-uncased'
tokenizer = BertTokenizer.from_pretrained(model_name)
model = BertModel.from_pretrained(model_name)

# Example sentence
sentence = "The cat sat on the mat."

# Tokenize the input sentence and convert tokens to tensor
input_ids = tokenizer.encode(sentence, return_tensors='pt')

# Pass the input through the BERT model to get embeddings
with torch.no_grad():
    outputs = model(input_ids)
    last_hidden_states = outputs.last_hidden_state

# Print the shape of the last hidden states tensor
print("Shape of last hidden states:", last_hidden_states.shape)

# Convert the embeddings to numpy array (for easier manipulation)
embeddings = last_hidden_states.squeeze().numpy()

# Tokenize the sentence to match embeddings to words
```

```
tokens = tokenizer.convert_ids_to_tokens(input_ids.squeeze())

# Print the tokens and their corresponding contextual embeddings
for token, embedding in zip(tokens, embeddings):
    print(f"Token: {token}")
    print(f"Embedding: {embedding[:10]}...")  # Print first 10 dimensions for b
    print()
```

Output:

```
Shape of last hidden states: torch.Size([1, 9, 768])
Token: [CLS]
Embedding: [-0.3642237  -0.05305378 -0.36732262 -0.02967339 -0.460784   -0.1010
  0.01669817  0.59577715 -0.11770311  0.10289837]...

Token: the
Embedding: [-0.3978658  -0.27210808 -0.68196577 -0.00734524  0.7860015   0.1766
  0.05241349  0.72017133  0.07858636 -0.1736162 ]...

Token: cat
Embedding: [-0.35117194 -0.07356024 -0.06913986 -0.13987705  0.68294847  0.1135
  0.20849192  0.56738263  0.4069492  -0.2134663 ]...

Token: sat
Embedding: [ 0.07117955 -0.31366652  0.09802647  0.06934201  0.4834015  -0.4046
 -0.5492254   0.91489977 -0.19875513  0.16641603]...

Token: on
Embedding: [-0.5203689  -0.59298277  0.28364897  0.31230223  0.611251   -0.0707
 -1.1455988   0.3248083  -0.40707844 -0.04888151]...

Token: the
Embedding: [-0.46198413 -0.5197541  -0.37599826  0.5099069   0.47716403 -0.4171
 -0.4499631   0.41355488 -0.52844054 -0.38209906]...

Token: mat
Embedding: [-0.0415443  -0.10548864 -0.28080556  0.5944824   0.05494812 -0.3332
  0.23721729  0.21435769 -0.5872034  -0.5192848 ]...

Token: .
Embedding: [-0.23536152 -0.4874898  -0.16314735  0.24718559  0.16603808 -0.1089
 -0.47729397  0.72053766 -0.12877737 -0.6664553 ]...

Token: [SEP]
```

```
Embedding: [ 0.66511077  0.02249792 -0.41309452  0.34166738 -0.2383636  -0.4008
  0.6143277   0.11614177  0.33811757  0.20712788]...
```

4.3 Modeling

After data is preprocessed, it is fed into an NLP architecture that models the data to accomplish a variety of tasks.

- Numerical features extracted by the techniques described above can be fed into various models depending on the task at hand. For example, for classification, the output from the TF-IDF vectorizer could be provided to logistic regression, naive Bayes, decision trees, or gradient boosted trees. Or, for named entity recognition, we can use hidden Markov models along with n-grams.
- Deep neural networks typically work without using extracted features, although we can still use TF-IDF or Bag-of-Words features as an input.
- Language Models: In very basic terms, the objective of a language model is to predict the next word when given a stream of input words. Probabilistic models that use Markov assumption are one example:

$$P(W_n) = P(W_n \mid W_{n-1})$$

Deep learning is also used to create such language models. Deep-learning models take as input a word embedding and, at each time state, return the probability distribution of the next word as the probability for every word in the dictionary. Pre-trained language models learn the structure of a particular language by processing a large corpus, such as Wikipedia. They can then be fine-tuned for a particular task. For instance, BERT has been fine-tuned for tasks ranging from fact-checking to writing headlines.

4.3.1 Named Entity Recognition (NER)

Definition: The process of identifying and classifying named entities in text into predefined categories such as person names, organizations, locations, dates, etc.

Example: In the sentence “Barack Obama was the 44th President of the United States,” “Barack Obama” is recognized as a person, and “United States” as a GPE (Geopolitical Entity)).

```
import spacy

# Load the pre-trained NLP model from spacy
nlp = spacy.load('en_core_web_sm')

# The sentence for which we want to perform NER
sentence = "Barack Obama was the 44th President of the United States."

# Process the sentence using the NLP model
doc = nlp(sentence)

# Print the named entities recognized in the sentence
print("Named Entities in the sentence:")
for ent in doc.ents:
    print(f"{ent.text}: {ent.label_}")
```

Output:

```
Named Entities in the sentence:
Barack Obama: PERSON
44th: ORDINAL
the United States: GPE
```

4.3.2 Language Model

Definition: A language model is a type of statistical model that predicts the next word in a sequence of words. It assigns probabilities to sequences of words, helping to determine the likelihood of a given sequence. Language models are fundamental in various natural language processing (NLP) tasks such as text generation, machine translation, and speech recognition.

Types:

- **Unigram Model:** Predicts each word independently based on its probability.
- **Bigram/Trigram Model:** Predicts a word based on the previous one or two words.
- **Neural Language Models:** Use neural networks, like Recurrent Neural Networks (RNNs) or Transformers, to capture more complex patterns and dependencies in text.

Importance: Fundamental in tasks such as text generation, machine translation, and speech recognition.

```
import nltk
from collections import defaultdict, Counter
import random

# Sample text data (corpus)
corpus = [
    "I love natural language processing",
    "I love machine learning",
    "I enjoy learning new things",
    "Natural language processing is fascinating"
]

# Tokenize the sentences into words
tokenized_corpus = [nltk.word_tokenize(sentence.lower()) for sentence in corpus

# Create bigrams from the tokenized corpus
bigrams = []
for sentence in tokenized_corpus:
    bigrams.extend(list(nltk.bigrams(sentence)))

# Calculate bigram frequencies
bigram_freq = defaultdict(Counter)
for w1, w2 in bigrams:
    bigram_freq[w1][w2] += 1

# Calculate bigram probabilities
bigram_prob = defaultdict(dict)
for w1 in bigram_freq:
    total_count = float(sum(bigram_freq[w1].values()))
    for w2 in bigram_freq[w1]:
        bigram_prob[w1][w2] = bigram_freq[w1][w2] / total_count

# Function to generate text using the bigram model
def generate_sentence(start_word, num_words=10):
    current_word = start_word
    sentence = [current_word]
    for _ in range(num_words - 1):
        if current_word in bigram_prob:
            next_word = random.choices(
                list(bigram_prob[current_word].keys()),
                list(bigram_prob[current_word].values())
            )[0]
            sentence.append(next_word)
            current_word = next_word
        else:
            break
    return ' '.join(sentence)
```

```
# Generate a sentence starting with the word "i"
generated_sentence = generate_sentence("i", num_words=4)
print("Generated sentence:", generated_sentence)
```

Output:

```
Generated sentence: i love natural language
```

4.3.3 Traditional ML NLP Techniques:

- Logistic regression is a supervised classification algorithm that aims to predict the probability that an event will occur based on some input. In NLP, logistic regression models can be applied to solve problems such as sentiment analysis, spam detection, and toxicity classification.
- Naive Bayes is a supervised classification algorithm that finds the conditional probability distribution *P(label | text)* using the following Bayes formula:

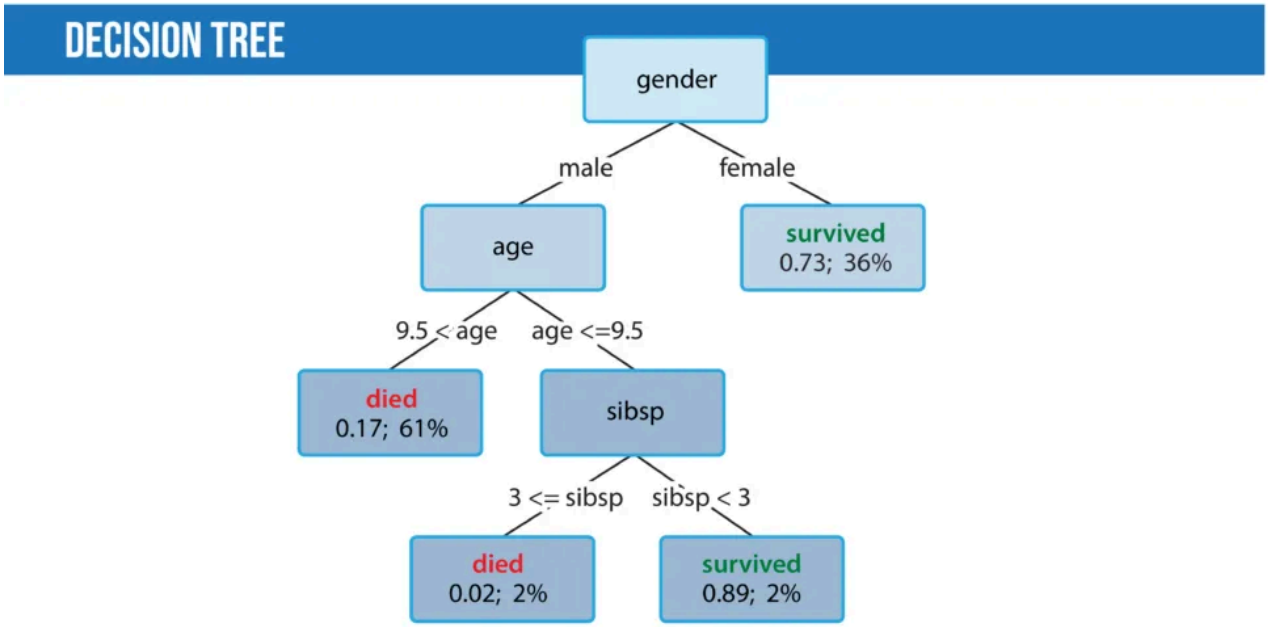
$$\mathrm{P(label \mid text)} = \frac{\mathrm{P(label)} \times \mathrm{P(text \mid label)}}{\mathrm{P(text)}}$$

and predicts based on which joint distribution has the highest probability. The naive assumption in the Naive Bayes model is that the individual words are independent. Thus:

$$\mathrm{P(text \mid label)} = \mathrm{P(word_1 \mid label)} \times \mathrm{P(word_2 \mid label)} \times \cdots \times \mathrm{P(word_n \mid label)}$$

In NLP, such statistical methods can be applied to solve problems such as spam detection or finding bugs in software code.

- Decision trees are a class of supervised classification models that split the dataset based on different features to maximize information gain in those splits.



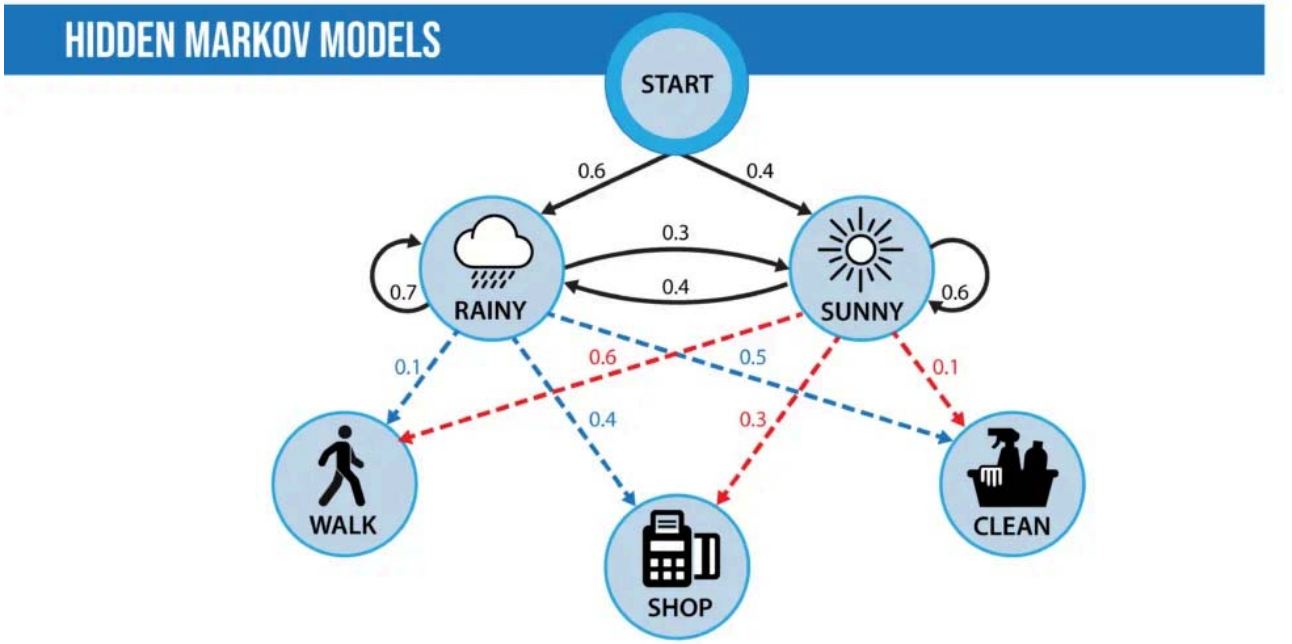
This decision tree assesses the survival of males and females aboard the Titanic when the ship sank. "Sibsp" stands for the number of siblings and spouses. The numbers of each leaf of the tree show first the probability of survival and second the percentage of examples classified by each leaf. Females had a 73 percent chance of survival, and young males with less than three siblings had a 89 percent chance of survival.

Credits: DeepLearning.AI

- **Latent Dirichlet Allocation (LDA)** is used for topic modeling. LDA tries to view a document as a collection of topics and a topic as a collection of words. LDA is a statistical approach. The intuition behind it is that we can describe any topic using only a small set of words from the corpus.

- **Hidden Markov models:** Markov models are probabilistic models that decide the next state of a system based on the current state. For example, in NLP, we might suggest the next word based on the previous word. We can model this as a Markov model where we might find the transition probabilities of going from word1 to word2, that is, P(word1|word2). Then we can use a product of these transition probabilities to find the probability of a sentence. The hidden Markov model (HMM) is a probabilistic modeling technique that introduces a hidden state to the Markov model. A hidden state is a property of the data that isn't directly observed. HMMs are used for part-of-speech (POS) tagging where the words of a sentence are the observed states and the POS tags are the hidden states. The HMM adds a concept called emission probability; the probability of an observation given a hidden state. In the prior example, this is the probability of a word, given its POS tag. HMMs assume that this probability can be reversed: Given a sentence, we can calculate the part-of-speech tag from each word based on both how likely a word was to have a certain part-of-speech tag and the probability that a particular part-of-speech tag follows the part-of-speech tag

assigned to the previous word. In practice, this is solved using the Viterbi algorithm.

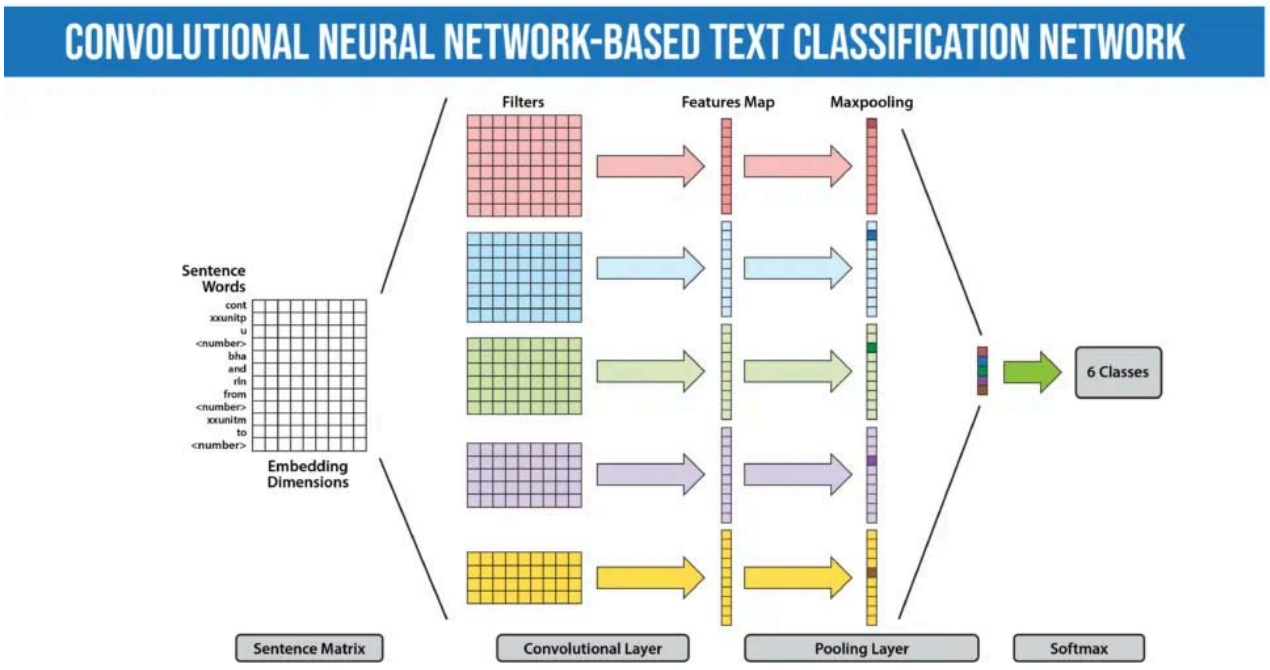


Someone is trying to recall the weather of last week based on what they did last week. Their actions are the observed states, and the different types of weather are the hidden states. Lines between weather states show the transition probabilities, while lines from weather states to actions show the emission probabilities (likelihood that they performed one action given the weather).

Credits: DeepLearning.AI

4.3.4 Deep Learning NLP Techniques

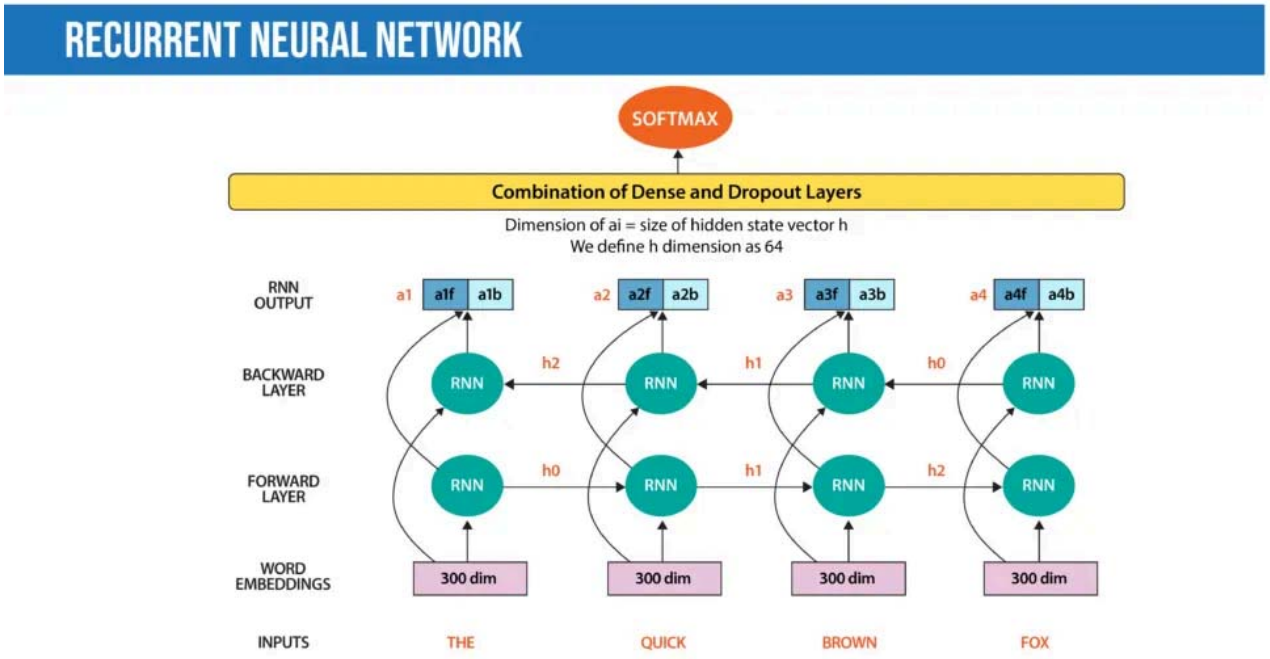
Convolutional Neural Network (CNN): The idea of using a CNN to classify text was first presented in the paper "Convolutional Neural Networks for Sentence Classification" by Yoon Kim. The central intuition is to see a document as an image. However, instead of pixels, the input is sentences or documents represented as a matrix of words.



Given a sentence, a convolutional neural network uses convolutional layers to refine representations of input words, before combining them to render a classification.

Credits: DeepLearning.AI

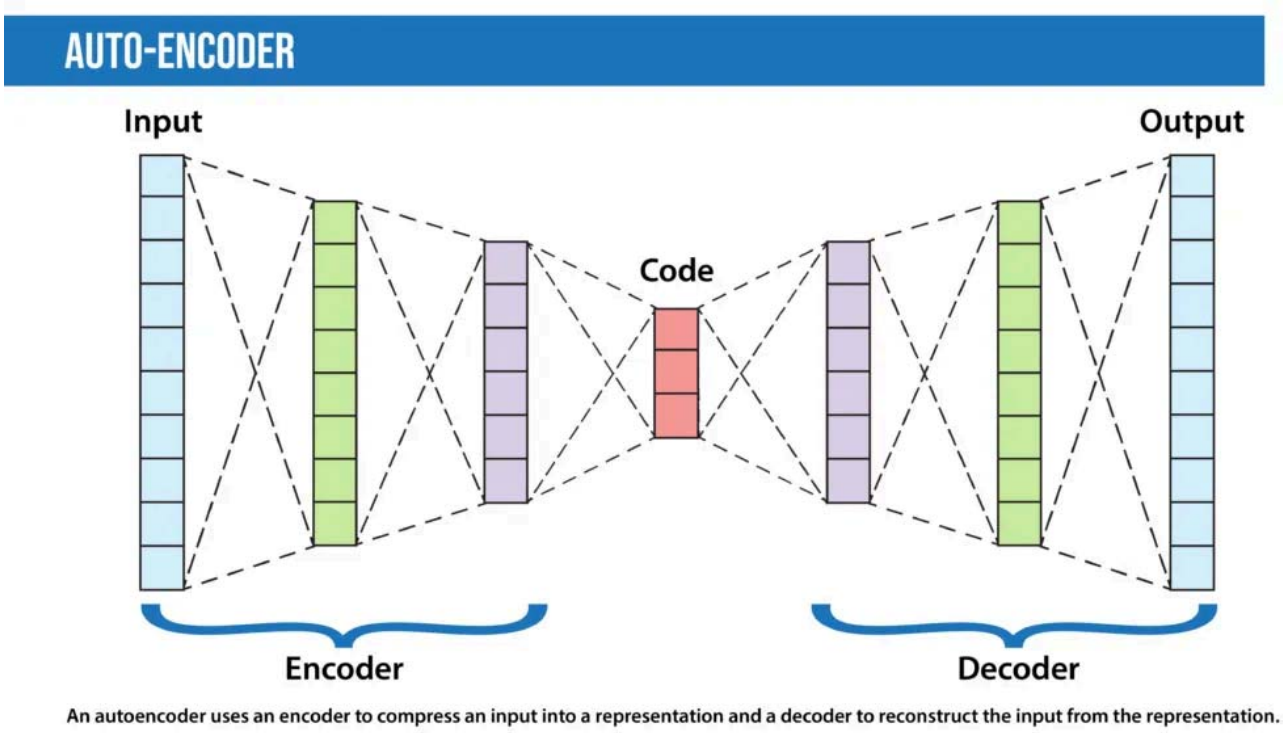
Recurrent Neural Network (RNN): Many techniques for text classification that use deep learning process words in close proximity using n-grams or a window (CNNs). They can see “New York” as a single instance. However, they can’t capture the context provided by a particular text sequence. They don’t learn the sequential structure of the data, where every word is dependent on the previous word or a word in the previous sentence. RNNs remember previous information using hidden states and connect it to the current task. The architectures known as Gated Recurrent Unit (GRU) and long short-term memory (LSTM) are types of RNNs designed to remember information for an extended period. Moreover, the bidirectional LSTM/GRU keeps contextual information in both directions, which is helpful in text classification. RNNs have also been used to generate mathematical proofs and translate human thoughts into words.



A bidirectional recurrent neural network processes the input both forward and backward to improve the representations it produces.

Credits: DeepLearning.AI

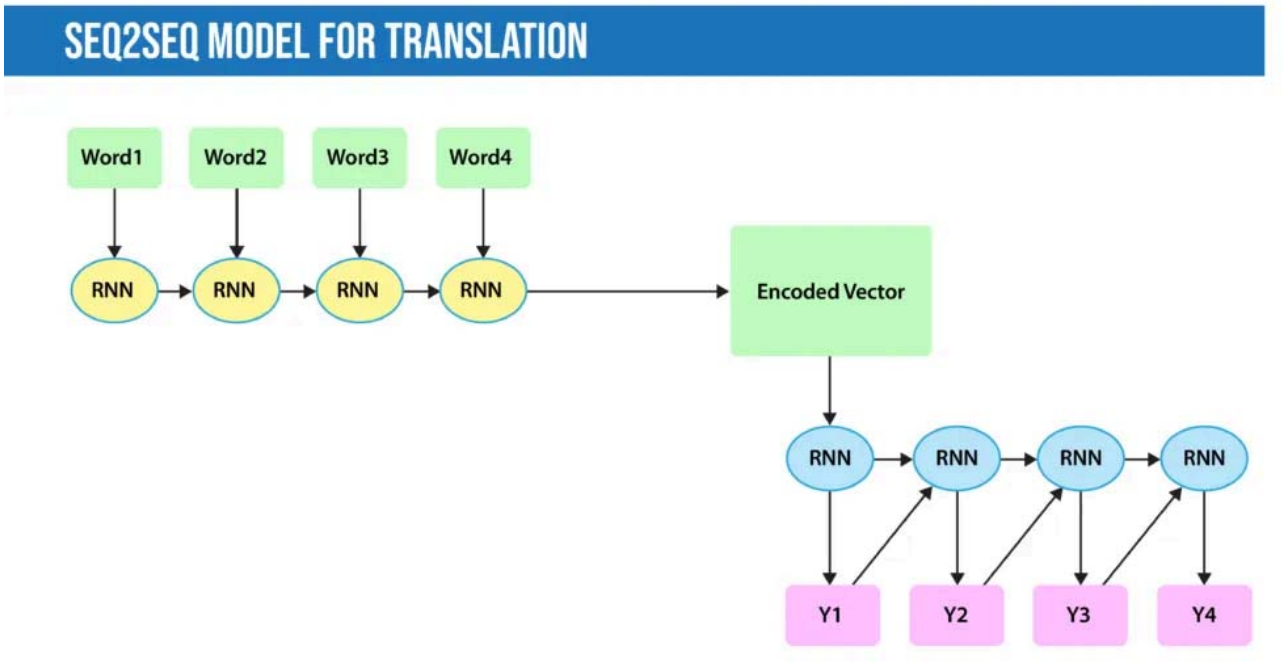
Autoencoders: These are deep learning encoder-decoders that approximate a mapping from X to X, i.e., input=output. They first compress the input features into a lower-dimensional representation (sometimes called a latent code, latent vector, or latent representation) and learn to reconstruct the input. The representation vector can be used as input to a separate model, so this technique can be used for dimensionality reduction. Among specialists in many other fields, geneticists have applied autoencoders to spot mutations associated with diseases in amino acid sequences.



An autoencoder uses an encoder to compress an input into a representation and a decoder to reconstruct the input from the representation.

Credits: DeepLearning.AI

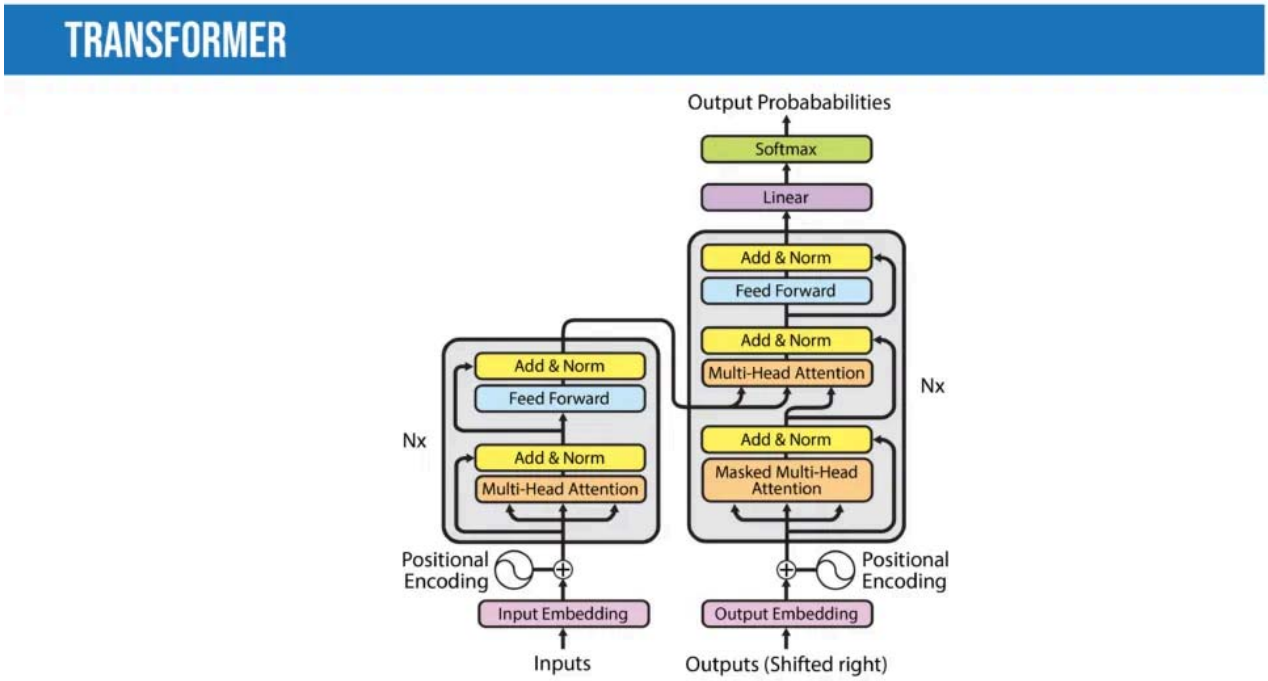
Encoder-decoder sequence-to-sequence: The encoder-decoder seq2seq architecture is an adaptation to autoencoders specialized for translation, summarization, and similar tasks. The encoder encapsulates the information in a text into an encoded vector. Unlike an autoencoder, instead of reconstructing the input from the encoded vector, the decoder's task is to generate a different desired output, like a translation or summary.



Given a sentence, a Recurrent Neural Network encodes the sentence and then iteratively generates a translation.

Credits: DeepLearning.AI

Transformers: The transformer, a model architecture first described in the 2017 paper "Attention Is All You Need" (Vaswani, Shazeer, Parmar, et al.), forgoes recurrence and instead relies entirely on a self-attention mechanism to draw global dependencies between input and output. Since this mechanism processes all words at once (instead of one at a time) that decreases training speed and inference cost compared to RNNs, especially since it is parallelizable. The transformer architecture has revolutionized NLP in recent years, leading to models including BLOOM, Jurassic-X, and Turing-NLG. It has also been successfully applied to a variety of different vision tasks, including making 3D images.



The encoder-decoder transformer used for translation. Encoder on the left, decoder on the right. Note that the decoder takes in its previously generated words during generation.

Credits: DeepLearning.AI

Key Concepts of the Transformer Model

Self-Attention Mechanism:

- Self-attention allows the model to focus on different parts of the input sequence when generating each part of the output. It computes a weighted sum of input features, with the weights determined by the relevance of each feature to others.

Multi-Head Attention:

- The model computes attention multiple times with different weight matrices (heads) and then concatenates the results. This allows the model to capture

different aspects of the relationships between words.

Positional Encoding:

- Since Transformers don't have a built-in sense of word order (unlike RNNs), positional encodings are added to the input embeddings to provide information about the position of each word in the sequence.

Feedforward Neural Network:

- After the attention mechanism, the data passes through a feedforward neural network, which is applied to each position separately and identically.

Layer Normalization and Residual Connections:

- Each sub-layer in the Transformer model is connected using residual connections followed by layer normalization, which helps stabilize the training of deep models.
- **Importance:** State-of-the-art in many NLP tasks due to its ability to capture long-range dependencies and parallelize computations.

```
import torch
import torch.nn as nn
import math

class TransformerModel(nn.Module):
    def __init__(self, input_dim, model_dim, num_heads, num_layers, ffn_dim, ma
        super(TransformerModel, self).__init__()

        self.embedding = nn.Embedding(input_dim, model_dim)
        self.model_dim = model_dim
        self.layers = nn.ModuleList([
            TransformerLayer(model_dim, num_heads, ffn_dim)
            for _ in range(num_layers)
        ])
        self.fc_out = nn.Linear(model_dim, num_classes)

    def forward(self, x):
        seq_len = x.size(1)
        positional_encoding = self._generate_positional_encoding(seq_len)
        x = self.embedding(x) + positional_encoding
        for layer in self.layers:
            x = layer(x)
```

```
            return self.fc_out(x.mean(dim=1))

        def _generate_positional_encoding(self, seq_len):
            positional_encoding = torch.zeros(seq_len, self.model_dim)
            for pos in range(seq_len):
                for i in range(0, self.model_dim, 2):
                    positional_encoding[pos, i] = math.sin(pos / (10000 ** (i / sel
                    positional_encoding[pos, i + 1] = math.cos(pos / (10000 ** ((i
            return positional_encoding.unsqueeze(0)

    class TransformerLayer(nn.Module):
        def __init__(self, model_dim, num_heads, ffn_dim):
            super(TransformerLayer, self).__init__()
            self.multihead_attention = nn.MultiheadAttention(embed_dim=model_dim, r
            self.norm1 = nn.LayerNorm(model_dim)
            self.ffn = nn.Sequential(
                nn.Linear(model_dim, ffn_dim),
                nn.ReLU(),
                nn.Linear(ffn_dim, model_dim)
            )
            self.norm2 = nn.LayerNorm(model_dim)

        def forward(self, x):
            attn_output, _ = self.multihead_attention(x, x, x)
            x = self.norm1(x + attn_output)
            ffn_output = self.ffn(x)
            x = self.norm2(x + ffn_output)
            return x

    # Example usage:
    input_dim = 10000  # Vocabulary size
    model_dim = 512
    num_heads = 8
    num_layers = 6
    ffn_dim = 2048
    max_seq_len = 100
    num_classes = 10

    # Instantiate the model
    model = TransformerModel(input_dim, model_dim, num_heads, num_layers, ffn_dim,

    # Example input (batch_size=32, sequence_length=50)
    x = torch.randint(0, input_dim, (32, 50))

    # Forward pass
    output = model(x)
    print("Output shape:", output.shape)
```

Output:

```
Output shape: torch.Size([32, 10])
```

4.3.5 Attention Mechanism

- **Definition:** A technique that allows models to focus on different parts of the input sequence when making predictions.
- **Importance:** Improves performance in sequence-to-sequence tasks by providing contextually relevant information.

4.3.6 Sequence-to-Sequence (Seq2Seq) Model

- **Definition:** A model architecture used for tasks where the input and output are sequences, such as machine translation.
- **Components:** Typically includes an encoder and a decoder, often using RNNs, LSTMs, or transformers.
- **Importance:** Used in tasks like translation, summarization, and dialogue systems.

4.3.7 Transfer Learning

- **Definition:** The technique of using pre-trained models on new, related tasks with minimal additional training.
- **Importance:** Allows leveraging existing knowledge, reducing the need for large annotated datasets and extensive training.

4.3.8 Fine-Tuning

- **Definition:** The process of taking a pre-trained model and adapting it to a specific task using additional training data.
- **Importance:** Enhances the performance of pre-trained models on task-specific applications.

4.3.9 Zero-Shot Learning

- **Definition:** The ability of a model to perform tasks it was not explicitly trained for, using general knowledge.
- **Importance:** Demonstrates the model's ability to generalize and adapt to new tasks without task-specific training data.

4.3.10 Few-Shot Learning

- **Definition:** The ability of a model to learn from a very small number of examples.

- **Importance:** Reduces the need for large amounts of annotated data and shows the model's ability to quickly adapt to new tasks.

5. Comparative Analysis of NLP Models and Techniques

Let's analyze each model and technique in comparison with its predecessor, highlighting their advantages and limitations:

- **Bag of Words (BoW) vs. TF-IDF:** BoW counts the occurrences of words in a document, while TF-IDF assigns weights to words based on their importance within the document and the entire corpus. TF-IDF addresses a key limitation of BoW by giving more importance to rare words, thereby capturing the document's meaning more effectively. However, both methods fail to consider word order and context, resulting in a loss of semantic information.

- **Word2Vec vs. BoW and TF-IDF:** Word2Vec, a neural network-based technique, learns continuous word embeddings that capture semantic relationships between words. Unlike BoW and TF-IDF, Word2Vec preserves contextual information and represents words in a dense vector space, allowing it to capture semantic relationships like synonyms, antonyms, and analogies. However, Word2Vec has its limitations, such as difficulties with out-of-vocabulary (OOV) words and the inability to capture different meanings of a word based on context (polysemy).

- **RNN (including LSTM and GRU) vs. Word2Vec:** While Word2Vec focuses on word representations, RNNs (Recurrent Neural Networks) are designed for modeling sequences of data, including text. RNNs can handle input sequences of varying lengths and maintain a hidden state that captures information from previous time steps. This makes RNNs more suitable than Word2Vec for tasks involving temporal dependencies, such as sentiment analysis or machine translation.

- **LSTM and GRU vs. Vanilla RNN:** Vanilla RNNs suffer from the vanishing gradient problem, which hampers their ability to learn long-range dependencies. LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit) are variants of RNNs designed to overcome this limitation by incorporating gating mechanisms that effectively capture long-term

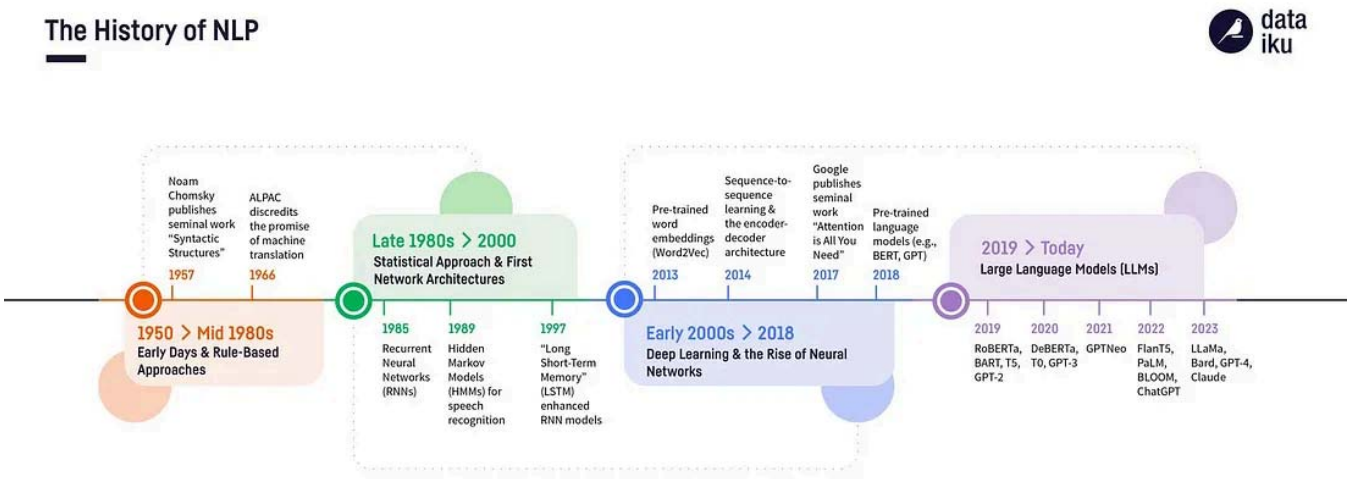
dependencies. However, RNNs, including LSTMs and GRUs, can be computationally expensive, particularly for long sequences.

- **Bi-directional LSTM vs. LSTM**: Bi-directional LSTMs extend traditional LSTMs by processing input sequences in both forward and backward directions. This allows the model to capture information from both past and future contexts, often leading to improved performance in tasks like named entity recognition and sentiment analysis. However, this approach is more computationally intensive than standard LSTMs due to the additional backward pass.

- **Transformer vs. RNN (LSTM, GRU)**: Transformers, a type of neural network architecture, leverage self-attention mechanisms to model dependencies between words in a sequence. Unlike RNNs, transformers can process input sequences in parallel, making them more computationally efficient, especially for long sequences. Additionally, transformers are more effective at capturing long-range dependencies, as they are not constrained by sequence length like RNNs. However, transformers can be memory-intensive due to the self-attention mechanism and often require large amounts of training data to achieve optimal performance.

6. The Evolution of NLP

Till now we have built a good understanding of NLP. Let's look into NLP evolution to understand the complete picture of the evolution of NLP.
We will be discussing each of them in detail in upcoming blogs.



Credits: dataiku

Natural Language Processing (NLP) has seen remarkable transformation since its inception in the 1950s. This journey can be categorized into distinct eras, each marked by groundbreaking advancements and innovations. Let's delve into the evolution of NLP, from its early rule-based systems to the sophisticated large language models (LLMs) we see today.

6.1 1950s to Mid-1980s: Early Days & Rule-Based Approaches

NLP's origins trace back to the 1950s, focusing on rule-based methods. During this period, researchers aimed to encode linguistic rules into computer programs, hoping to enable machines to understand and generate human language.

Key Milestones

- **1957:** Noam Chomsky's seminal work, *Syntactic Structures*, introduced transformational grammar, profoundly influencing computational linguistics.
- **1966**: The ALPAC report discredited early machine translation efforts, highlighting the challenges and complexities of language processing.

6.2 Late 1980s to 2000: Statistical Approaches & First Network Architectures

The late 1980s marked a shift from rule-based to statistical methods, thanks to increasing computational power and the availability of large datasets.

Key Milestones

- **1985:** The introduction of Recurrent Neural Networks (RNNs) laid the groundwork for sequence-based data processing. *Learning Representations by Back-Propagating Errors*
- **1989**: Hidden Markov Models (HMMs) became popular for speech recognition and other sequence-based tasks. *Hidden Markov Models for Speech Recognition*
- **1997:** Long Short-Term Memory (LSTM) networks, developed by Hochreiter and Schmidhuber, improved RNNs by addressing the vanishing gradient problem. *Long Short-Term Memory*

Development of Statistical Methods

- **Bag of Words (BoW)**: A foundational method that represents text data by counting word occurrences, ignoring grammar and word order.
- **TF-IDF (Term Frequency-Inverse Document Frequency)**: An enhancement over BoW that weighs word frequency inversely with its document occurrence, highlighting rarer, informative words. *Introduction to Information Retrieval*

6.3 Early 2000s to 2018: Deep Learning & the Rise of Neural Networks

The early 2000s heralded the rise of deep learning techniques, leveraging large neural networks trained on vast datasets to revolutionize NLP.

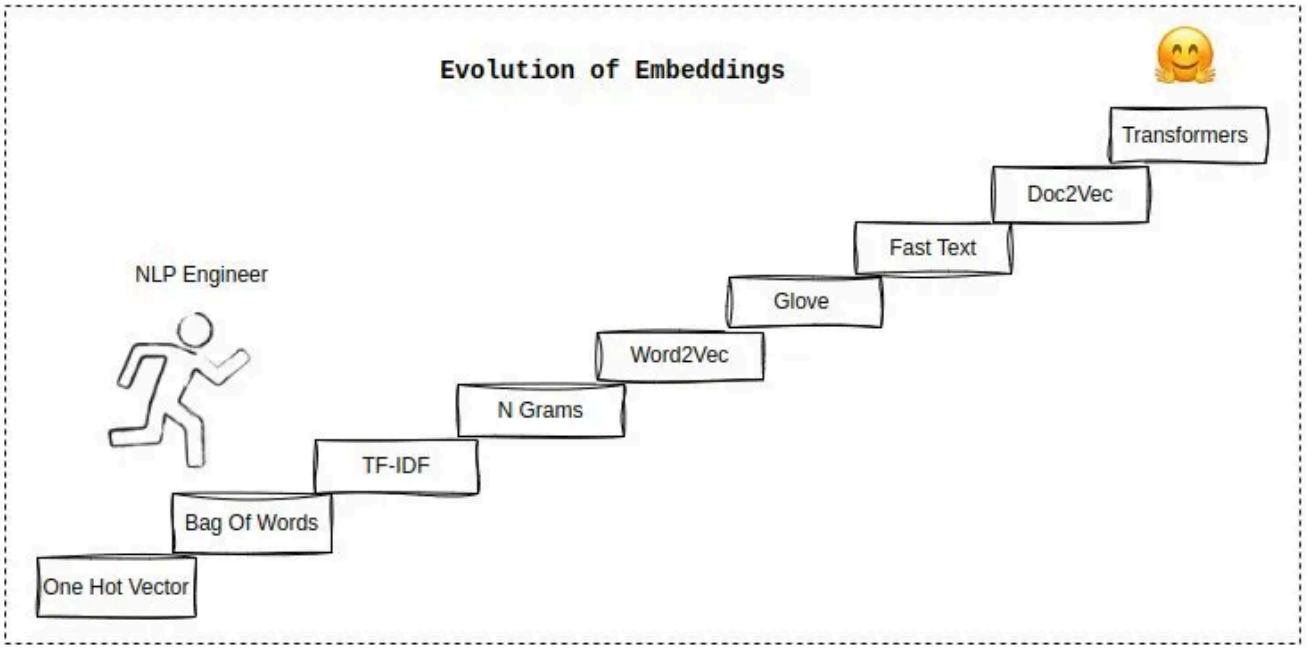
Key Milestones

- **2013:** Word2Vec, introduced by Mikolov et al., enabled the creation of dense word embeddings, capturing semantic relationships between words. *Efficient Estimation of Word Representations in Vector Space*
- **2014:** Sequence-to-sequence learning and the encoder-decoder architecture, introduced by Sutskever, Vinyals, and Le, significantly advanced machine translation. *Sequence to Sequence Learning with Neural Networks*
- **2014:** Bahdanau, Cho, and Bengio introduced the attention mechanism, which allowed models to focus on specific parts of the input sequence during translation. *Neural Machine Translation by Jointly Learning to Align and Translate*
- **2017:** Google's Transformer model, introduced in the paper *Attention is All You Need*, revolutionized NLP by relying entirely on attention mechanisms.

Detailed Development of Word Embedding Methods

Word embeddings transformed how machines understand words. The journey began with Word2Vec, a neural network that processes text by "vectorizing" words, placing semantically similar words close together in a continuous vector space.

- **2013:** Word2Vec transformed word embeddings using Continuous Bag-of-Words (CBOW) and Skip-Gram architectures. *Efficient Estimation of Word Representations in Vector Space*
- **2014:** GloVe, developed by Pennington et al. at Stanford, aggregated global word-word co-occurrence statistics from a corpus. *GloVe: Global Vectors for Word Representation*
- **2018:** FastText, from Facebook's AI Research lab, improved upon Word2Vec by considering subword information, making it effective for morphologically rich languages. *Enriching Word Vectors with Subword Information*



Credits: Ketan Goel

Attention Mechanisms

- **2014:** Bahdanau et al. introduced the attention mechanism, significantly improving translation by allowing models to focus on relevant parts of the input sequence. *Neural Machine Translation by Jointly Learning to Align and Translate*
- **2017:** Vaswani et al. introduced the Transformer model, which uses attention mechanisms exclusively, improving efficiency and performance in NLP tasks. *Attention is All You Need*

Embedding Techniques in Transformers

Transformers introduced novel embedding techniques, including position-wise feed-forward networks and positional encodings, helping models understand word order in sentences.*Attention is All You Need*

6.4 2019 to Today: Large Language Models (LLMs)

In recent years, large language models (LLMs) have pushed the boundaries of NLP capabilities.

Key Milestones

2019: Models like RoBERTa, BART, T5, and GPT-2 set new benchmarks for language understanding and generation.

- *RoBERTa: A Robustly Optimized BERT Pretraining Approach*

- *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*
- *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*
- *Language Models are Unsupervised Multitask Learners*

2020: OpenAI's GPT-3, with 175 billion parameters, demonstrated impressive language generation and understanding capabilities. *Language Models are Few-Shot Learners*

2021: Advancements continued with models like DeBERTa, T0, and the open-source GPT-Neo.

- DeBERTa: Decoding-enhanced BERT with Disentangled Attention
- Multitask Prompted Training Enables Zero-Shot Task Generalization
- *GPT-Neo*

2022: Models like FlanT5, PaLM, BLOOM, and ChatGPT brought nuanced and context-aware language understanding to the forefront.

- *Scaling Instruction-Finetuned Language Models*
- *PaLM: Scaling Language Modeling with Pathways*
- *BLOOM: A 176B-parameter Open-Access Multilingual Language Model*
- *ChatGPT*

2023: The development of models like LLaMa, Bard, GPT-4, and Claude demonstrated ongoing progress in NLP.

- *LLaMa: Large Language Model Meta AI*
- *GPT-4*
- Bard and Claude are proprietary models, with details on their respective company websites.

Fine-Tuning of Large Language Models

Fine-tuning is essential in adapting pre-trained LLMs to specific tasks. This involves

additional training on smaller, task-specific datasets, allowing the model to specialize while leveraging its vast pre-trained knowledge.

- **2019:** Techniques like supervised and unsupervised fine-tuning with models such as BERT and RoBERTa became prevalent.
- **2020:** GPT-3 highlighted the potential of few-shot learning, where models are fine-tuned with just a few examples to achieve impressive performance on diverse tasks. *Language Models are Few-Shot Learners*

The history of NLP showcases the incredible strides made over the past seven decades. From early rule-based systems to the sophisticated large language models of today, each era has significantly contributed to our understanding of human language. As we continue to develop more advanced models, the future of NLP promises even more exciting innovations and applications, bringing us closer to truly understanding and generating human language.

7. Conclusion

In this blog, we have embarked on a journey to understand the fundamentals of Natural Language Processing (NLP). We began by defining NLP and exploring its core areas: Natural Language Understanding (NLU) and Natural Language Generation (NLG). We then delved into various applications of NLP, including sentiment analysis, machine translation, and named entity recognition, highlighting how these tasks are tackled using different techniques and models.

We also covered essential NLP concepts and terminologies such as tokenization, stemming, lemmatization, and the role of features like Bag of Words (BoW) and TF-IDF in feature extraction. Additionally, we discussed advanced techniques like word embeddings and transformers, which have revolutionized the field by enabling more sophisticated language understanding and generation.

Finally, we explored the evolution of NLP, tracing its development from rule-based systems in the 1950s to the powerful large language models (LLMs) of today, such as GPT-3 and GPT-4. This historical context provided a comprehensive overview of how far the field has come and the significant milestones that have shaped modern NLP.

By understanding these foundational concepts and the evolution of NLP, readers are well-equipped to further explore the intricacies of NLP in future discussions and applications.

8. Test your Knowledge!

8.1 Basic Interview Questions

1. What are the two main areas of NLP, and how do they differ?
— Expected Answer: Natural Language Understanding (NLU) focuses on interpreting the meaning behind text, while Natural Language Generation (NLG) deals with producing human-like text.

2. Explain the difference between stemming and lemmatization. Why might one be preferred over the other in certain NLP tasks?
— Expected Answer: Stemming reduces words to their root form by stripping affixes, which can result in non-words. Lemmatization, on the other hand, reduces words to their base or dictionary form (lemma) and generally produces real words. Lemmatization is often preferred when accuracy is critical, while stemming is faster and less computationally expensive.

3. What is the purpose of tokenization in NLP, and what are the different types?
— Expected Answer: Tokenization breaks down text into smaller units called tokens, which can be words, subwords, or characters. Types include word tokenization, sentence tokenization, and subword tokenization.

4. How does Term Frequency-Inverse Document Frequency (TF-IDF) differ from the Bag of Words (BoW) model, and what problem does it solve?
— Expected Answer: TF-IDF improves upon BoW by not only counting word occurrences but also weighting words based on their importance in the document relative to the entire corpus. It addresses the problem of common words being overrepresented by reducing their importance.

5. What are word embeddings, and how do they improve upon traditional models like BoW and TF-IDF?
— Expected Answer: Word embeddings are dense vector representations of words that capture their meanings, syntactic properties, and relationships with other words. Unlike BoW and TF-IDF, which treat words as discrete entities, word embeddings place semantically similar words close together in a continuous vector space, preserving context.

6. Describe the Transformer model and its significance in NLP. How does it differ from previous models like RNNs?
— Expected Answer: The Transformer model relies entirely on self-attention

mechanisms to handle sequential data without processing it in order, making it more efficient and capable of capturing long-range dependencies compared to RNNs, which process data sequentially and are less efficient for long sequences.

7. What role does the attention mechanism play in modern NLP models, and how does it improve the performance of tasks like translation?
— Expected Answer: The attention mechanism allows models to focus on specific parts of the input sequence when generating each part of the output, providing contextually relevant information and improving the performance of tasks like translation by focusing on relevant words or phrases.

8. What is the difference between fine-tuning and zero-shot learning in the context of large language models?
— Expected Answer: Fine-tuning involves adapting a pre-trained model to a specific task using additional training data, while zero-shot learning refers to a model's ability to perform tasks it wasn't explicitly trained for by using general knowledge.

9. How has the evolution of NLP from rule-based systems to large language models (LLMs) changed the field?
— Expected Answer: The evolution from rule-based systems to LLMs has transformed NLP by moving from manually coded linguistic rules to data-driven approaches that leverage massive datasets and advanced architectures like Transformers, resulting in models capable of more sophisticated language understanding and generation.

10. In the context of NLP, what is a corpus, and how is it used in model training?
— Expected Answer: A corpus is a large collection of documents that serve as the dataset on which NLP models are trained and evaluated. It is used to analyze linguistic patterns and build statistical models.

8.2 Advanced Questions

1. Given a scenario where your NLP model is failing to correctly classify sentiments in customer reviews, what steps would you take to diagnose and improve the model?
— Expected Answer: Steps might include analyzing the data for class imbalance, reviewing the preprocessing pipeline (e.g., tokenization, stop-word removal), checking the feature extraction method (e.g., TF-IDF vs. word embeddings), exploring alternative models (e.g., using transformers instead of simple logistic

regression), and validating the performance on a different validation set to rule out overfitting.

2. How would you handle out-of-vocabulary (OOV) words in a production NLP system?
— Expected Answer: Approaches could include using subword tokenization methods like Byte Pair Encoding (BPE) or using character-level embeddings. Another method might be to replace OOV words with a special token and fine-tune the model to handle these cases better.

3. Imagine you are tasked with building a named entity recognition (NER) system for a legal document corpus. What challenges might you face, and how would you address them?
— Expected Answer: Challenges could include dealing with domain-specific language, long and complex sentences, and the need for high accuracy due to legal implications. To address these, one might use pre-trained models fine-tuned on legal text, incorporate domain-specific embeddings, and apply techniques like active learning to iteratively improve the model's performance.

4. If your chatbot is providing irrelevant answers to user queries, how would you debug the issue?
— Expected Answer: The debugging process might involve checking the intent recognition system, validating the dialogue flow and context management, ensuring the training data is diverse and representative, and analyzing logs to see where the conversation deviated. You might also inspect the model's confidence scores to identify if the issue is with the model's understanding or with the response generation logic.

5. How would you design an NLP system to automatically generate summaries of research papers? What factors would you consider?
— Expected Answer: Considerations would include choosing between extractive vs. abstractive summarization methods, evaluating the quality and length of summaries, handling domain-specific terminology, ensuring that the model captures the key contributions of the paper, and addressing challenges like redundancy and coherence. You might also discuss the importance of fine-tuning pre-trained models on a dataset of research paper summaries.

6. Suppose you are working with a text classification model that uses TF-IDF features. How might you incorporate additional information, such as the sentiment of the text or the author's demographic information, to improve performance?
— Expected Answer: One approach could involve creating additional features for the sentiment score or author demographics and concatenating them with the TF-IDF vectors before feeding them into the classifier. Another approach could be to use a multi-input model where one branch processes the TF-IDF features and another processes the additional information, combining the outputs before classification.

7. What strategies would you employ to reduce model bias in an NLP system designed for hiring purposes?
— Expected Answer: Strategies might include careful selection of training data to ensure diversity, using techniques like adversarial debiasing, monitoring the model for biased outputs, and implementing fairness constraints during training. It might also be important to conduct regular audits of the model's decisions and to include a human-in-the-loop for sensitive decisions.

8. How would you explain the concept of the attention mechanism in transformers to someone without a technical background?
— Expected Answer: You might explain that the attention mechanism allows the model to focus on important words in a sentence, similar to how a person might pay attention to certain words when trying to understand a complex sentence. For instance, in the sentence "The cat sat on the mat," the model would pay more attention to "cat" and "mat" to understand the main idea.

9. You are asked to optimize the performance of a machine translation model that struggles with rare words. What solutions would you propose?
— Expected Answer: Solutions could include implementing subword tokenization to break down rare words into more common subword units, using transfer learning from a model pre-trained on a large corpus, fine-tuning the model on a dataset that includes rare words, or augmenting the dataset with paraphrases or synthetic data that includes these rare words.

10. In a scenario where your NLP model's predictions are difficult to interpret, how would you go about improving the interpretability of the model?
— Expected Answer: Approaches could include using simpler models like decision trees or logistic regression, implementing attention visualization in transformer

models to see which parts of the input the model focuses on, using techniques like LIME or SHAP to explain predictions, and incorporating post-hoc analysis to break down how the model arrived at a particular decision.

8.3 DIY

1. What are some practical applications of NLP?
2. How do we represent word embeddings ?
3. Why do we need CNN in NLP ?
4. What is transfer learning? What are the steps you would take to perform transfer learning?
5. What is the main difference between GPT and BERT series?
6. If there were no libraries like NLTK & SpaCy, what would be your approach to build Named Entity Recognition pipeline?
7. How will you decide the position of the words in text?
8. How are transformers beneficial over other models like RNNs?
9. Is text processing possible with CNN? Why is it not preferred and RNNs more preferred for it?
10. What is a recurrent neural network (RNN), and how does it handle sequential data in NLP?
11. What is the difference between abstractive and extractive summarisation?

Credits

In this blog post, we have compiled information from various sources, including research papers, technical blogs, official documentations, YouTube videos, and more. Each source has been appropriately credited beneath the corresponding images, with source links provided.

Below is a consolidated list of references:

1. https://www.xoriant.com/blog/natural-language-processing-the-next-disruptive-technology-under-ai-part-i
2. https://datasciencedojo.com/blog/natural-language-processing-applications/

3. https://www.deeplearning.ai/resources/natural-language-processing/
4. https://www.researchgate.net/figure/Natural-Language-Processing_fig1_364842359
5. https://www.linkedin.com/pulse/20141209180635-83626359-nlp-and-text-analytics-similified-document-classification/
6. https://geekflare.com/natural-language-understanding/
7. https://blog.dataiku.com/nlp-metamorphosis
8. https://www.linkedin.com/posts/ketan-gangal_embeddings-in-natural-language-processing-activity-7066631663706882048-LAWl/

Thank you for reading!

If this guide has enhanced your understanding of Python and Machine Learning:

- Please show your support with a clap 👏 or several claps!
- Your claps help me create more valuable content for our vibrant Python or ML community.
- Feel free to share this guide with fellow Python or AI / ML enthusiasts.
- Your feedback is invaluable — it inspires and guides my future posts.

Connect with me!

Vipra

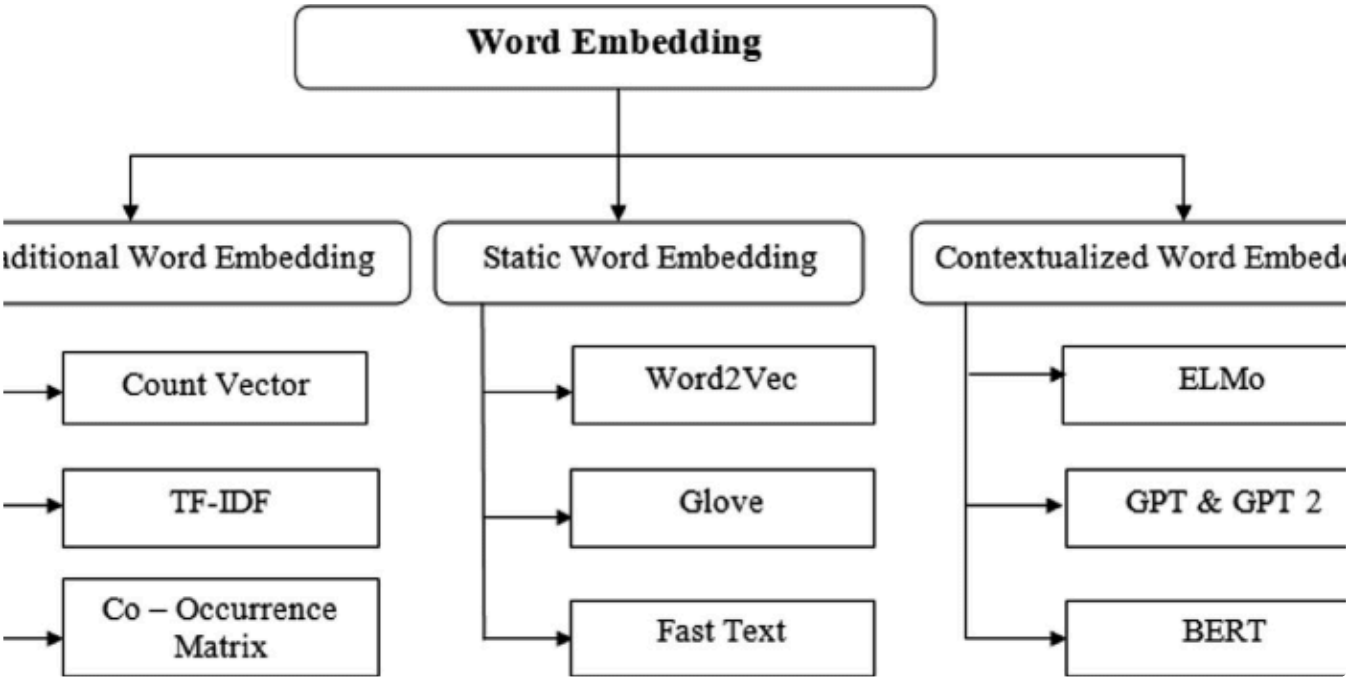
Naturallanguageprocessing | Large Language Models | Transformers | Embedding | Interview

Follow

Written by Vipra Singh

2K Followers

More from Vipra Singh

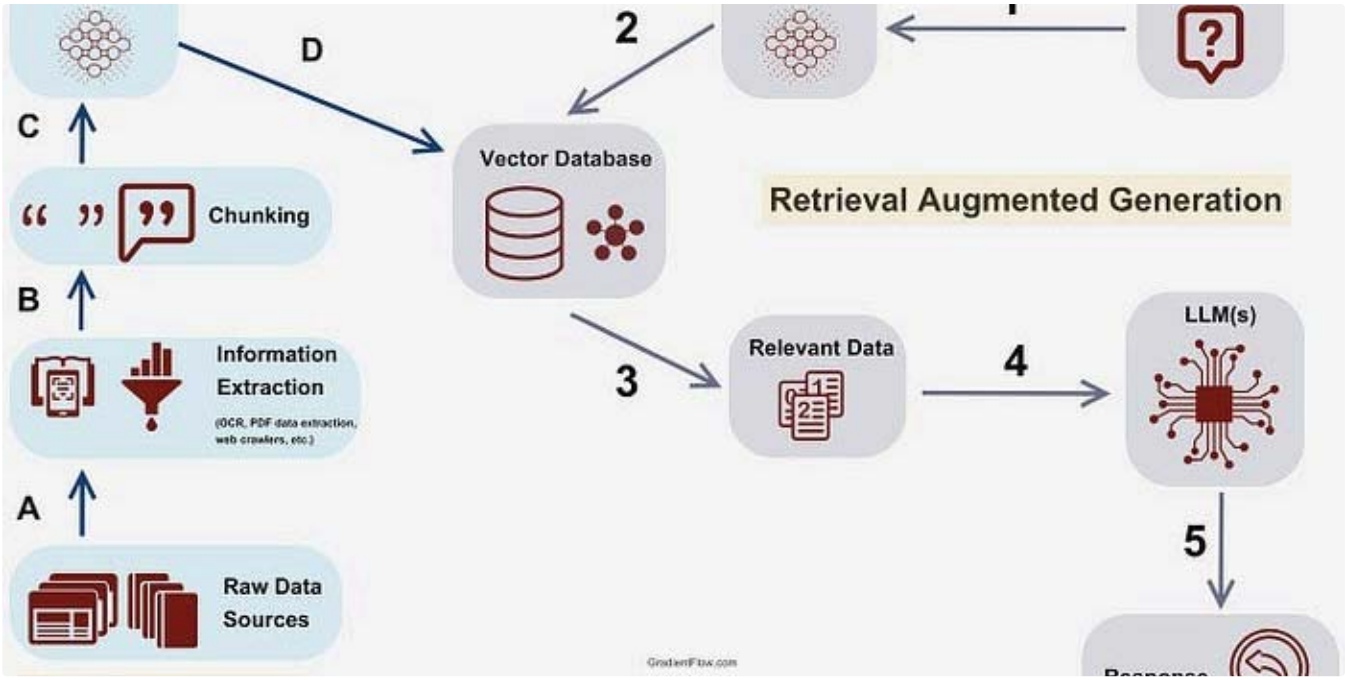


Vipra Singh

LLM Architectures Explained: Word Embeddings (Part 2)

Deep Dive into the architecture & building real-world applications leveraging NLP Models starting from RNN to Transformer.

Aug 18 138 3

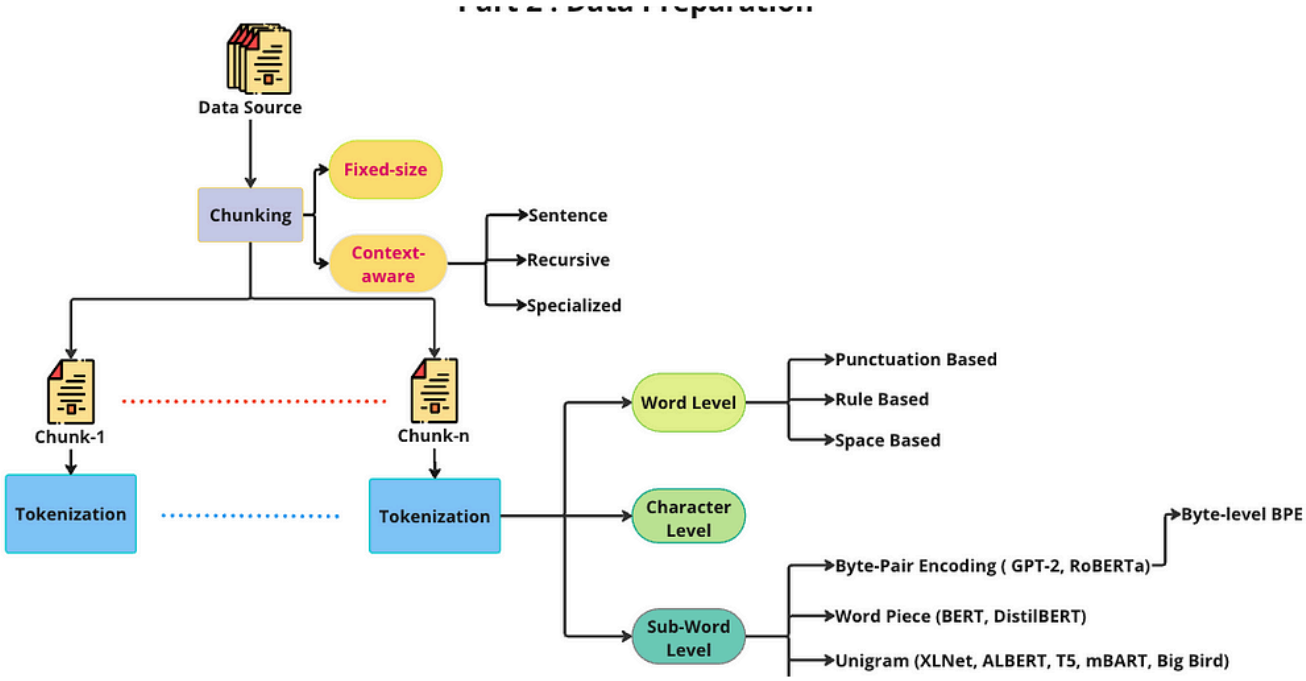


Vipra Singh

Building LLM Applications: Introduction (Part 1)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented Generation (RAG) Application.

Jan 9 1.1K 5

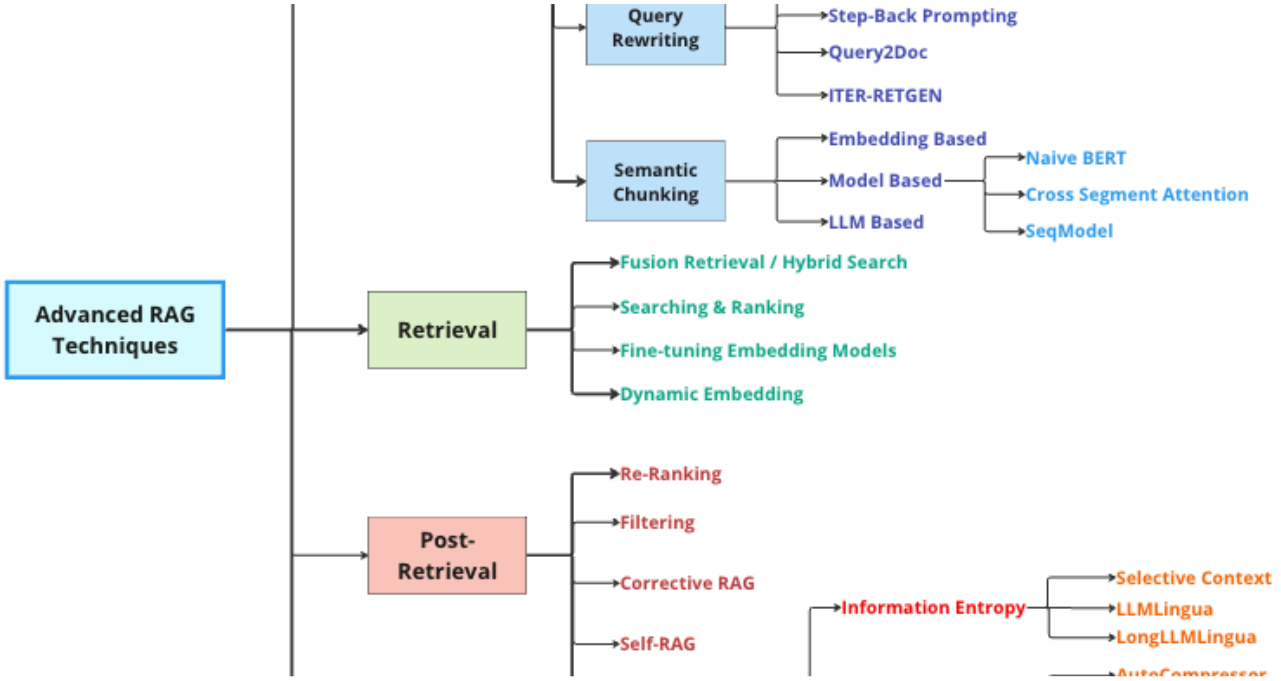


Vipra Singh

Building LLM Applications: Data Preparation (Part 2)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented Generation (RAG) Application.

Jan 9 596 2



Vipra Singh

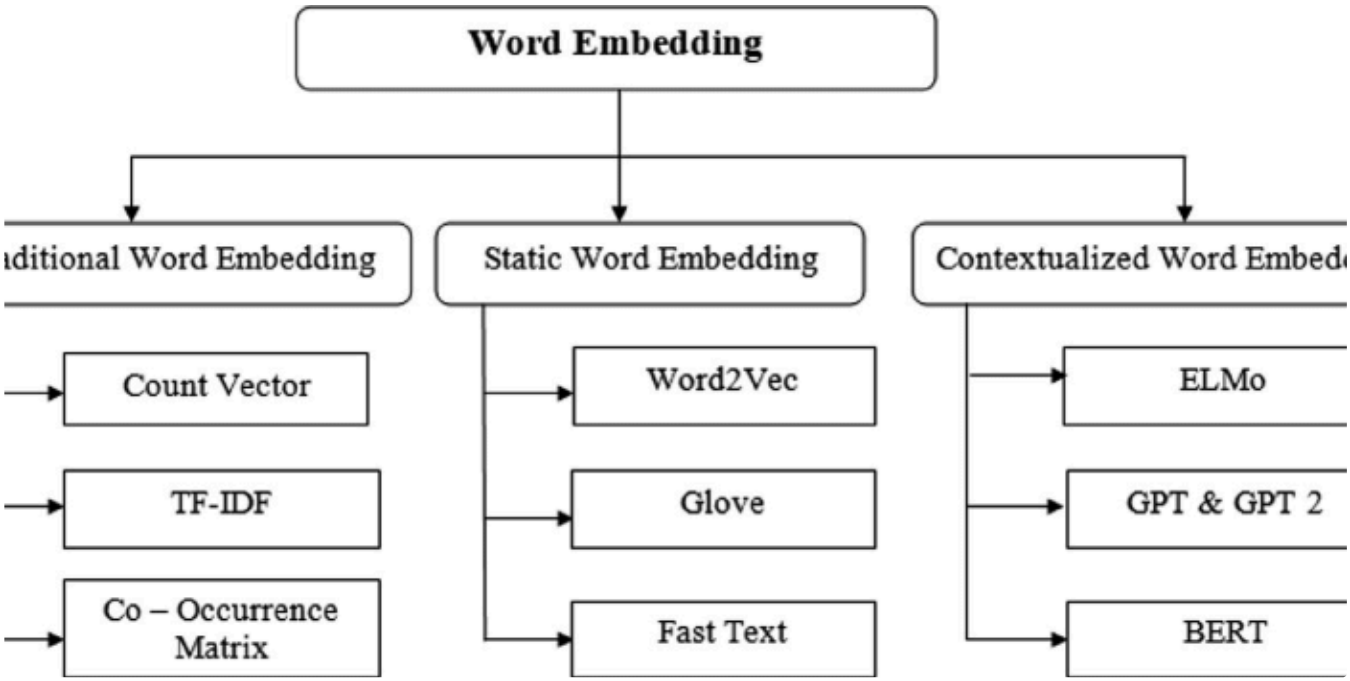
Building LLM Applications: Advanced RAG (Part 10)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented Generation (RAG) Application.

Apr 28 665 5

See all from Vipra Singh

Recommended from Medium

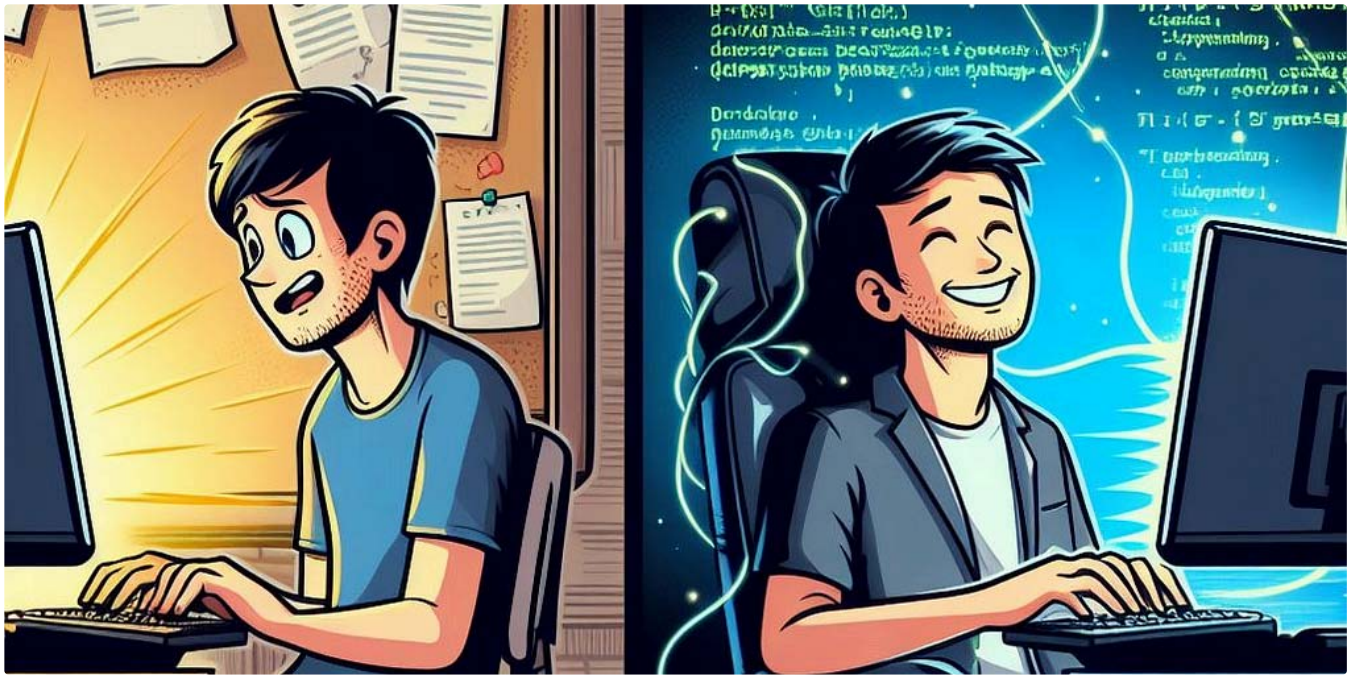


Vipra Singh

LLM Architectures Explained: Word Embeddings (Part 2)

Deep Dive into the architecture & building real-world applications leveraging NLP Models starting from RNN to Transformer.

Aug 18 138 3



Abhay Parashar in The Pythoneers

17 Mindblowing Python Automation Scripts I Use Everyday

Scripts That Increased My Productivity and Performance

Aug 25 7.3K 69

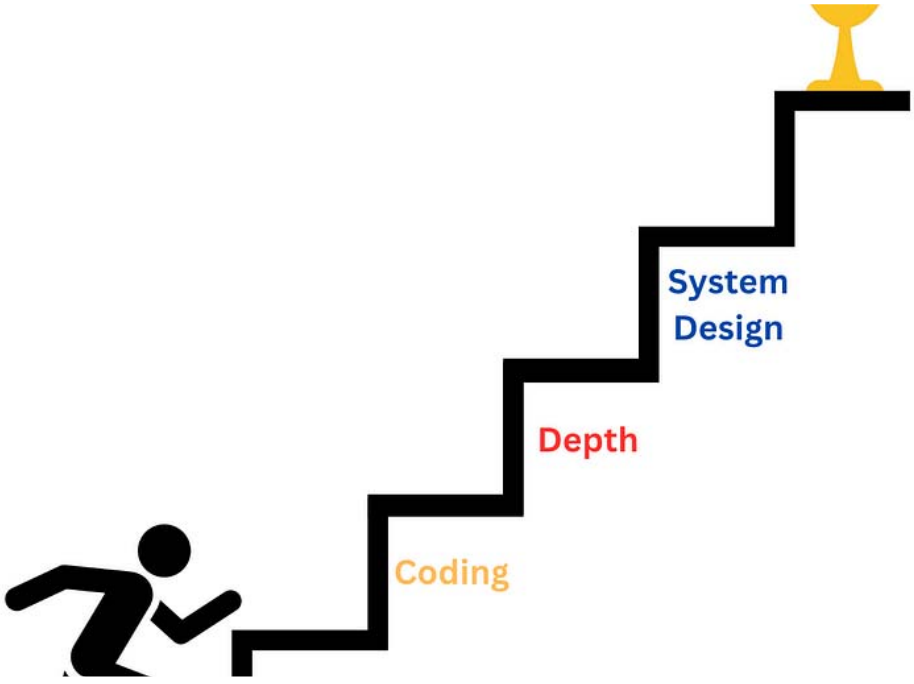
Lists

Natural Language Processing
1675 stories · 1252 saves

AI Regulation
6 stories · 557 saves

ChatGPT prompts
48 stories · 1953 saves

data science and AI
40 stories · 234 saves



Kartik Singhal

How to Ace Machine Learning Interviews: My Personal Playbook

Resources that helped me prepare for ML interview rounds

Jul 31 542 2

Andrew Best in Artificial Intelligence in Plain English

Why OpenAI's "Strawberry" is a GAME CHANGER!

This is a big step closer to AGI.

Aug 9 1.4K 33

Valentina Alto

Introducing Agent-based RAG

An implementation with LangGraph, Azure AI Search and Azure OpenAI GPT-4o

Aug 16 276 3

Benedict Neo in bitgrit Data Science Publication

Forget `pip install`, Use This Instead

Install Python packages up to 100x ⚡ faster than before.

Mar 27 2.3K 23

See more recommendations