Table of Contents

Data Archive Book

Welcome to the Data Archive Book.

Preface

Remember the Data Archive is non-linear — use internal links for navigation. If you need a specific page see the **Table of Contents** to jump to a topic.

Asking Questions

Why Ask Better Questions?

Asking better questions enhances thinking, learning, problem-solving, and communication. Good questions:

- Guide inquiry and exploration.
- Clarify assumptions.
- Elicit insights or novel responses.
- Enable self-reflection and deeper understanding.

What Makes a Good Question?

A good question:

- **Elicits a novel or thoughtful response** — not just a fact or yes/no.
- **Opens possibilities** rather than closing them.
- **Reveals assumptions** or **forces re-evaluation** of mental models.
- **Matches the context and audience** — good questions for brainstorming differ from those for debugging.
- **Fosters chain-of-thought reasoning**, helping others articulate how they arrive at conclusions.

Types of Questions

Questions can be classified by their **function**, **depth**, or **structure**.

1. By Function

Type	Purpose	Example
Clarifying	Understand what's being said	"What do you mean by X?"
Probing	Dig deeper into reasoning or logic	"Why do you think that?"
Exploratory	Generate ideas or new perspectives	"What if we reversed the problem?"
Reflective	Encourage self-awareness	"What assumption am I making here?"
Critical	Test or challenge statements	"What evidence supports that?"

2. By Depth

- **Surface-level:** "What is X?"
- **Mid-level:** "How does X relate to Y?"
- **Deep-level:** "Why does X matter?" or "What are the implications of X?"

3. By Structure

- **Open-ended:** Encourage elaboration.

  → *"How might we design this differently?"*

- **Closed:** Seek a specific answer.

  → *"Is this implementation correct?"*

- **Leading:** Suggest an answer.

  → *"Wouldn't you agree that…?"*

- **Falsifiable/Testable:** Can be proven right or wrong.

  → *"Does increasing X always decrease Y?"*

Characteristics of Good Questions

Characteristic	Description
Purposeful	Serves a clear goal in the conversation or inquiry
Contextual	Relevant to the topic or the respondent
Open/Expansive	Invites multiple viewpoints or lines of reasoning
Challenging	Pushes beyond defaults or surface-level answers
Precise	Minimizes ambiguity while leaving room for elaboration
Sequenced	Ordered to build thought step-by-step (chain of thought)

Related Concepts

- Chain of Thought
- Design Thinking Questions
- Prompting

Questions:

- How LLMs generate or refine questions using Prompting or Chain of thought approaches?

Bag Of Words

In ML_Tools see: Bag_of_Words.py

In the context of natural language processing (NLP), the Bag of Words (BoW) model is a simple and commonly used method for text representation. It converts text data into numerical form by treating each document as a collection of individual words, disregarding grammar and word order. Here's how it works:

1. Vocabulary Creation: A vocabulary is created from the entire corpus, which is a list of all unique words appearing in the documents.
2. Vector Representation: Each document is represented as a vector, where each element corresponds to a word in the vocabulary. The value of each element is typically the count of occurrences of the word in the document.
3. Simplicity and Limitations: While BoW is easy to implement and useful for tasks like text classification, it has limitations. It ignores word order and context, and can result in large, sparse vectors for large vocabularies.

Despite its simplicity, BoW can be effective for certain NLP tasks, especially when combined with other techniques like TF-IDF to weigh the importance of words.

Takes key terms of a text in normalised unordered form.

`CountVectorizer` from scikit-learn to convert a collection of text documents into a matrix of token counts.

```
#Need normalize_document
from sklearn.feature_extraction.text import CountVectorizer


# Using CountVectorizer with the custom tokenizer
bow = CountVectorizer(tokenizer=normalize_document)
bow.fit(corpus)  # Fitting text to this model
print(bow.get_feature_names_out())  # Key terms
```

Represent each sentence by a vector of length determined by get_feature_names_out. representing the tokens contained.

Bert

BERT (Bidirectional Encoder Representations from Transformer) is used in NLPprocessing, developed by Google.

Introduced in the paper "BERT Pretraining of Deep Bidirectional Transformers for Language Understanding" in 2018.

It is forward & backward looking in the context.

BERT is a stack of encoders -learning context.

Input Vector Embedding|embedding:

- Positional Encoding: passes location info to encoder
- Sentence embeddings: differences between sentences
- Token embeddings

Training of BERT:

- Masked Language modelling (hiding words)

- Next Sentence Prediction

Fine tuning (Transfer Learning) BERT model:

- New output layer dependent

Resources:

- What is BERT and how does it work? | A Quick Review

What is BERT?

- BERT is based on the Transformer architecture and utilizes a bidirectional approach, meaning it considers the context of a word based on both its left and right surroundings in a sentence. This allows BERT to capture nuanced meanings and relationships between words more effectively than unidirectional models
- Pre-training and Fine-tuning/Transfer Learning techniques. It learns to predict masked words in sentences (Masked Language Model) and to determine if one sentence follows another (Next Sentence Prediction).

What is BERT Used For?

- Text Classification: Assigning categories or labels to text documents, such as sentiment analysis or topic classification.
- Named Entity Recognition (Named Entity Recognition|NER): Identifying and classifying entities (e.g., names, organizations, locations) within text.
- Question Answering: Providing answers to questions based on a given context or passage of text.
- Text Summarisation: Generating concise summaries of longer documents while retaining key information.
- Language Translation: Assisting in translating text from one language to another.
- Sentence Similarity :Measuring the similarity between sentences, which can be useful for tasks like paraphrase detection or duplicate question identification.

Embeddings For Oov Words

Can you find words in a Vector Embedding|word embedding that where not used to creates the embedding?

Yes, but with important caveats. If a word is not in the spaCy model's vocabulary with a vector, then:

✅ What you can do

Option 1: Filter out words without vectors (what you're doing now)

This is the cleanest option:

```
if token.has_vector:
    embeddings.append(token.vector)
    valid_words.append(word)
```

Option 2: Fallback to character-level embeddings (optional)

If you're using `en_core_web_lg`, spaCy sometimes provides approximate vectors for out-of-vocabulary (OOV) words using subword features. But with `en_core_web_md`, OOV words truly lack vector meaning.

Option 3: Use a different embedding model

Use FastText or transformer-based models (e.g., Sentence Transformers), which can produce embeddings for OOV words based on subword information or context.

Example with FastText (using gensim):

```
from gensim.models import KeyedVectors

model = KeyedVectors.load_word2vec_format("cc.en.300.vec")  # or download from FastText
embedding = model.get_vector("unseenword")  # FastText will synthesize it
```

💡 Summary

Approach	Handles OOV?	Notes
spaCy `en_core_web_md`	❌	Skips words without vectors (recommended)
spaCy `en_core_web_lg`	⚠️ Sometimes	May infer vectors using subword info
FastText / GloVe	✅	Good for unseen words
Sentence Transformers (BERT)	✅	Contextualized, ideal for phrases/sentences

NLP #ml_process #ml_optimisation

Faiss

FAISS (Facebook AI Similarity Search) is a library developed by Facebook AI Research that enables efficient similarity search and clustering of dense vectors. It is especially well-suited for applications involving high-dimensional vector data, such as NLP

Related terms:

- Vector Embedding

Overview

FAISS is optimized for:

- **Fast retrieval** from large collections of vectors (millions or more).
- **Approximate nearest neighbor (ANN)** search, which trades off accuracy for speed.
- **Exact search**, depending on the chosen index type.
- **GPU acceleration** for very large-scale search tasks.

Core Concept

At its core, FAISS takes a large number of **high-dimensional vectors** (e.g., sentence or document embeddings), and enables fast **similarity search** to retrieve the most similar vectors to a given Querying|query.

For example, in an NLP Memory|context:

- Documents or notes are embedded into vector space using a model like SBERT.
- These embeddings are stored in a FAISS index.
- Given a query, its embedding is computed, and FAISS returns the nearest neighbors (i.e., most semantically similar notes).

Index Types

FAISS offers different types of indices depending on use case:

- `IndexFlatL2` : exact search using L2 (Euclidean) distance.
- `IndexIVFFlat` : approximate search using inverted files.
- `IndexHNSW` : Hierarchical Navigable Small World graph-based index (good for high recall).
- `IndexPQ` : product quantization for memory-efficient indexing.

Lemmatization

Lemmatization is the process of reducing a word to its base or root form, known as the "lemma."

Unlike stemming, which simply cuts off word endings, lemmatization considers the context and morphological analysis of the words.

It ensures that the root word is a valid word in the language. For example, the words "running," "ran," and "runs" would all be lemmatized to "run."

This process helps in normalizing text data for natural language processing tasks by grouping together different forms of a word.

Named Entity Recognition

Named Entity Recognition (NER) is a subtask of NLP|Natural Language Processing (NLP) that involves identifying and classifying key entities in text into predefined categories such as names, organizations, locations.

The process typically employs algorithms like Conditional Random Fields (CRFs) or deep learning models such as Bi-directional LSTM (Long Short-Term Memory) networks.

Mathematically, NER can be framed as a sequence labeling problem where the goal is to assign a label y_i to each token x_i in a sentence. The model learns from annotated datasets, optimizing parameters to maximize the likelihood $P(y|x)$ using techniques like backpropagation.

NER has significant implications in information extraction, search engines, and automated customer support systems.

Important

- NER transforms unstructured text into structured data for analysis.
- The choice of model significantly impacts the accuracy of entity recognition.

Example

An example of NER is identifying "Apple Inc." as an organization in the sentence: "Apple Inc. released a new product."

Follow up questions

- How does the choice of training data affect the performance of NER models
- What are the challenges of NER in multilingual contexts
- Why is named entity recognition (NER) a challenging task
- In NER how would you handle ambiguous entities

Related Topics

- Text classification in NLP
- Information extraction techniques

Nlp

Natural Language Processing (NLP) involves the interaction between computers and humans using natural language. It encompasses various techniques and models to process and analyze large amounts of natural language data.

Key Concepts

Preprocessing

- **Normalisation of Text**: The process of converting text into a standard format, which may include lowercasing, removing punctuation, and stemming or lemmatization.
- **Part of speech tagging**: Assigning a specific part-of-speech category (such as noun, verb, adjective, etc.) to each word in a text.

Models

- **Bag of words**: Represents text data by counting the occurrence of each word in a document, ignoring grammar and word order. It takes key terms of a text in normalized **unordered** form.
- **TF-IDF**: Stands for Term Frequency-Inverse Document Frequency. It improves on Bag of Words by considering the importance of a word in a document relative to its frequency across multiple documents.
- **Vectorization**: Converting text into numerical vectors. Techniques like Bag of Words, TF-IDF, or standardised/Vector Embedding (e.g., Word2Vec, GloVe) are used to represent text data numerically.

Analysis

- **One Hot Encoding**: Converts categorical data into a binary vector representation, indicating the presence or absence of a word from a list in the given text.

Methods

- **Ngrams**: Creates tokens from groupings of words, not just single words. Useful for capturing context and meaning in text data.

- **Grammar method**: Involves analyzing the grammatical structure of sentences to extract meaning and relationships between words.

Actions

- **Summarisation**: The process of distilling the most important information from a text to produce a concise version.

Tools and Libraries

General Imports

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


porter_stemmer = PorterStemmer()
wordnet_lemmatizer = WordNetLemmatizer()
```

- **nltk**: A leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources.
  - **punkt**: An unsupervised trainable model for tokenizing text into sentences and words.
  - **stopwords**: Commonly used words (such as "the", "is", "in") that are often removed from text data because they do not carry significant meaning.
  - **wordnet**: A lexical database for the English language that groups words into sets of synonyms and provides short definitions and usage examples.
  - **re**: Regular expressions for pattern matching and text manipulation.

Normalisation Of Text

Preprocessing in NLP tasks is called Normalization involves reducing words to their base or root form, converting them to lowercase, and removing stop words.

Processes

What are some steps involved in the pre-processing of a text?. These include making the text lower case, removal of punctuation, tokenize the text (split up the words in a sentence), remove stop words as they convey grammar rather than meaning, word stemming (reduce words to their stems).

Tokenisation: Used to separate words or sentences.

Stemming: returns part of a words that doesnt change ie breaks, breakthrough gives break. Use

```
from nltk.stem.porter import PorterStemmer
temp=text #decomposed
porter_stemmer = PorterStemmer()
stemmed_tokens = [porter_stemmer.stem(token) for token in temp]

print(stemmed_tokens)
```

lemmatization: reducing word to it's base form e.g. words "is", "was", "were" will turn into "be".

```
from nltk.stem.wordnet import WordNetLemmatizer
temp=text #decomposed
wordnet_lemmatizer = WordNetLemmatizer()
lemmatized_tokens = [wordnet_lemmatizer.lemmatize(token, pos="v") for token in temp]
print(lemmatized_tokens)
```

Code

main normaliser

```
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords

def normalize_document(document, stemmer=porter_stemmer, lemmatizer=wodnet_lemmatizer):
    """Noramlizes data by performing following steps:
        1. Changing each word in corpus to lowercase.
        2. Removing special characters and interpunction.
        3. Dividing text into tokens.
        4. Removing english stopwords.
        5. Stemming words.
        6. Lemmatizing words.
    """

    temp = document.lower()
    temp = re.sub(r"[^a-zA-Z0-9]", " ", temp)
    temp = word_tokenize(temp)
    temp = [t for t in temp if t not in stopwords.words("english")]
    temp = [porter_stemmer.stem(token) for token in temp]
    temp = [lemmatizer.lemmatize(token) for token in temp]
    return temp
```

Prompt Engineering

Prompt engineering is a technique in the field of natural language processing (NLP), particularly when working with large language models (LLMs).

It involves designing and optimizing input prompts to get the most relevant and accurate responses from these models.

Techniques like prompt retrievers, which include systems like UPRISE and DaSLaM, enhance the ability to retrieve and generate contextually appropriate prompts.

Prompt engineering aims to guide LLMs toward producing desired outputs while minimizing ambiguity.

Key Takeaways

- Prompt engineering optimizes input to improve LLM responses.
- Techniques like prompt retrievers (e.g., UPRISE, DaSLaM) enhance prompt effectiveness.
- Quality prompts reduce ambiguity and guide model outputs.
- Applications span multiple industries, enhancing user interaction and content generation.

Key Components Breakdown

Methods:

- **Prompt Design**: Crafting specific, clear prompts to guide model responses.
- **Prompt Retrieval**: Utilizing systems like UPRISE and DaSLaM to find effective prompts based on context.

Concepts:

- **Contextualization**: Understanding the context in which prompts are used to improve relevance.
- **Iterative Testing**: Continuously refining prompts based on model performance.

Algorithms:

- **Retrieval-Augmented Generation (RAG)**: Combines retrieval of relevant documents with generative responses.
- **Few-Shot Learning**: Providing examples within prompts to guide model behavior.

Concerns, Limitations, or Challenges

- **Ambiguity**: Poorly designed prompts can lead to vague or irrelevant responses.
- **Dependence on Training Data**: LLMs may produce biased or inaccurate outputs based on their training data.
- **Complexity**: Designing effective prompts requires a deep understanding of both the model and the task.

Example

For instance, if a user wants to generate a summary of a scientific article, a poorly constructed prompt like "Summarize this" may yield unsatisfactory results. In contrast, a well-engineered prompt such as "Provide a concise summary of the key findings and implications of the following article on climate change" is likely to produce a more relevant and informative response.

Follow-Up Questions

1. What are the best practices for evaluating the effectiveness of different prompts
2. How can prompt engineering be integrated into existing NLP workflows to enhance performance

Semantic Relationships

Semantic relationships refer to the connections and associations between words and concepts based on their meanings.

Understanding these relationships can enhance various natural language processing tasks, such as information retrieval, text analysis, and sentiment analysis.

Leveraging Lexical Resources like WordNet

One of the key resources for exploring semantic relationships is **WordNet**, a lexical database that groups words into sets of cognitive synonyms called **synsets**. These synsets are linked together in a hierarchy based on semantic relations, including:

- **Hypernymy**: Represents an "is-a" relationship (e.g., "dog" is a hypernym of "beagle").
- **Hyponymy**: Represents a more specific type (e.g., "beagle" is a hyponym of "dog").

You can use WordNet to find synonyms or related concepts for important words (those with high TF-IDF scores) in your documents. If different documents contain synonyms or words related in the WordNet hierarchy, this may indicate a semantic relationship between them, even if the exact words differ.

WordNet also provides measures of semantic similarity between synsets based on their paths in the hypernym hierarchy. These measures can be explored to quantify the semantic relatedness of key terms in your documents. The Natural Language Toolkit (NLTK) offers an interface to access WordNet.

Sentiment Analysis with SentiWordNet

Another valuable resource is **SentiWordNet**, which extends WordNet by assigning sentiment scores (positive, negative, objective) to different senses of words. While your primary goal may be to explore semantic relationships, analyzing the sentiment expressed in your documents based on important words can provide an additional layer of understanding.

Documents discussing similar topics might also share similar sentiments, strengthening the case for a semantic link. NLTK provides access to SentiWordNet, allowing you to incorporate sentiment analysis into your exploration of semantic relationships.

Small Language Models

LLM|LLMs dominate many general-purpose NLP tasks, small Language Models have their own place in specialized tasks, where they excel due to computational efficiency, interpretability, and task-specific fine-tuning.

SLMs remain highly relevant for Edge Machine Learning Models and edge computing, domain-specific tasks, and applications requiring interpretability, making them a crucial tool in the NLP landscape.

Use Cases for Small Language Models (SLMs)

- Contrastive Decoding: Improve the quality of generated content by filtering out low-quality outputs, by having a SLM guide and critique a LLM or other way (inference)
  - Mitigate hallucinations
  - Augmented Reasoning
- Distillation: Transfer the knowledge from a larger model to a smaller one, retaining performance but reducing computational requirements (see BERT Teacher model).

- Data Synthesis: Generate or augment datasets in scenarios with limited data.
- Model Cascading: Use a combination of smaller models and larger models in a cascading architecture, where simpler tasks are handled by SLMs and more complex ones by LLMs. Model cascading and routing allow SMs to handle simpler tasks, reducing computational overhead. Or the other way first do a general search with a LLM then refine to domain specific small model which is more interpretability|interpretable and specific.
- Domain specific & Limited Data Availability: SMs, however, can be effectively fine-tuned on smaller, domain-specific datasets and outperform general LLMs in tasks with limited data availability.
- RAG (Retrieval Augmented Generation): Lightweight retrievers (SMs) can support LLMs in finding relevant external information.

Advantages of SLMs

- Require less computational power and are faster in inference compared to LLMs.
- Interpretability
- Accessible for those without resources in power and data

Summarisation

Summarization in NLP

Summarization in natural language processing (NLP) is the process of condensing a text document into a shorter version while retaining its main ideas and key information. There are two primary forms of summarization:

The unsupervised summarization process involves splitting text, tokenizing sentences, assigning scores based on importance, and selecting top sentences. Effective scoring methods include calculating sentence similarity and analyzing word frequencies to ensure that the summary captures the essence of the original text.

Extraction:

- This method involves selecting specific words or sentences directly from the original text to create a summary. It focuses on identifying and pulling out the most important parts of the text without altering the original wording. Abstraction:
- The abstraction method generates a summary that may include new words and phrases not present in the original text. This approach is more complex as it requires understanding the content and rephrasing it, often using techniques like paraphrasing.

Unsupervised Summarization Process

The basic idea behind unsupervised summarization involves the following steps:

1. **Split Text into Sentences**: The text is divided into individual sentences for analysis.
2. **Tokenize Sentences**: Each sentence is tokenized into separate words, allowing for detailed examination of word usage.
3. **Assign Scores to Sentences**: Sentences are evaluated based on their importance, which is a crucial step in the summarization process.
4. **Select Top Sentences**: The highest-scoring sentences are selected and displayed in their original order to form the summary.

Methods for Assigning Scores

The main point of summarization is effectively assigning scores to sentences. Here are some common methods for doing this:

- **Similarity Calculation**: Calculate the similarity between each pair of sentences and select those that are most similar to the majority of sentences. This helps identify sentences that capture the central themes of the text.
- **Word Frequencies**: Analyze word frequencies to identify the most common words in the text. Sentences that contain a higher number of these frequent words are then selected for the summary.

Tf Idf

TF-IDF is a statistical technique used in text analysis to determine the importance of a word in a document relative to a collection of documents (corpus). It balances two ideas:

- Term Frequency (TF): Captures how often a term occurs in a document.
- Inverse Document Frequency (IDF): Discounts terms that appear in many documents.

High TF-IDF scores indicate terms that are frequent in a document but rare in the corpus, making them useful for distinguishing between documents in tasks such as information retrieval, document classification, and recommendation.

TF-IDF combines local and global term Statistics:

- TF gives high scores to frequent terms in a document
- IDF reduces the weight of common terms across documents
- TF-IDF identifies terms that are both frequent and distinctive

Equations

Term Frequency

$TF(t, d)$ measures how often a term t appears in a document d, normalized by the total number of terms in d:

$$TF(t,d) = \frac{f_{t,d}}{\sum_k f_{k,d}}$$

Where:

- $f_{t,d}$ is the raw count of term t in document d
- $\sum*k f*{k,d}$ is the total number of terms in d (i.e. the document length)

Inverse Document Frequency

IDF assigns lower weights to frequent terms:

$$IDF(t,D) = \log\left(\frac{N}{1 + |\{d \in D : t \in d\}|}\right)$$

Where:

- N is the number of documents in the corpus D
- $|{d \in D : t \in d}|$ is the number of documents containing term t
- Adding 1 to the denominator avoids division by zero

TF-IDF Score

The final score is:

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

Related Notes

- Bag of words
- Tokenisation
- Clustering
- Search
- Recommender systems
- nltk

Exploratory Ideas

- Can track TF-IDF over time (e.g., note evolution)
- Can cluster or classify the documents using TF-IDF?

Implementations

Python Script (scikit-learn version)

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer

# Step 1: Tokenize and vectorize using Bag of Words
bow = CountVectorizer(tokenizer=normalize_document)
X_counts = bow.fit_transform(corpus)

# Step 2: Apply TF-IDF transformation
tfidf_transformer = TfidfTransformer()
X_tfidf = tfidf_transformer.fit_transform(X_counts)

# Optional: View TF-IDF scores per document
for doc_id in range(len(corpus)):
    print(f"Document {doc_id}: {corpus[doc_id]}")
    print("TF-IDF values:")
    tfidf_vector = X_tfidf[doc_id].T.toarray()
    for term, score in zip(bow.get_feature_names_out(), tfidf_vector):
        if score > 0:
            print(f"{term.rjust(10)} : {score[0]:.4f}")
```

Python Script (custom TF-IDF implementation)

```
import math
from nltk.corpus import stopwords
from nltk.tokenize import RegexpTokenizer
from nltk.util import bigrams, trigrams

stop_words = stopwords.words('english')
tokenizer = RegexpTokenizer(r'\w+')

def tokenize(text):
    tokens = tokenizer.tokenize(text.lower())
    tokens = [t for t in tokens if len(t) > 2 and t not in stop_words]
    return tokens + [' '.join(b) for b in bigrams(tokens)] + [' '.join(t) for t in trigrams(tokens)]

def tf(term, doc_tokens):
    return doc_tokens.count(term) / len(doc_tokens)

def idf(term, docs_tokens):
    doc_count = sum(1 for doc in docs_tokens if term in doc)
    return math.log(len(docs_tokens) / (1 + doc_count))

def compute_tfidf(docs):
    docs_tokens = [tokenize(doc) for doc in docs]
    all_terms = set(term for doc in docs_tokens for term in doc)
    tfidf_scores = []
    for tokens in docs_tokens:
        tfidf = {}
        for term in all_terms:
            if term in tokens:
                tfidf[term] = tf(term, tokens) * idf(term, docs_tokens)
        tfidf_scores.append(tfidf)
    return tfidf_scores
```

Tokenisation

Tokenisation is a fundamental process in natural language processing (NLP) that involves breaking down text into smaller units called tokens. These tokens can be words, sentences, or subwords, depending on the level of tokenization.

Word tokenisation

```
from nltk.tokenize import word_tokenize #keeps punctuation
text_word_tokens_nltk = word_tokenize(text_original)
print(text_word_tokens_nltk)
```

Sentence tokenisation

```
from nltk.tokenize import sent_tokenize
text_sentence_tokens_nltk = sent_tokenize(text_original)
print(text_sentence_tokens_nltk)
```

Basic tokenisation

```
temp = text_original.lower()
temp = re.sub(r"[^a-zA-Z0-9]", " ", temp) # just letters and numbers
temp = re.sub(r"\[[0-9]+\]", "", temp) #remove weird stuff
temp = word_tokenize(temp) #break up text to word list
tokens_no_stopwords = [token for token in temp if token not in stopwords.words("english")] #remove common words
print(tokens_no_stopwords)
```

Transformer

A transformer in machine learning (ML) refers to a deep learning model architecture designed to process sequential data, such as natural language processing (NLP). It was introduced in the paper "standardised/Attention Is All You Need" and has since become a cornerstone in NLP tasks.

Transformers excel at handling sequence-based data and are particularly known for their self-attention mechanisms Attention mechanism, which allow them to process long-range dependencies in data.

Key Concepts of a Transformer

1. **Architecture Overview**:
    - A transformer model consists of an encoder and a decoder, although some models use only the **encoder** (like BERT only consists of encoders) or only the **decoder** (like GPT3). Each of these components is made up of layers that include mechanisms for attention and Feed Forward Neural Network.
    - Encoder learns the context, decoder does the task.
2. **Self-Attention Mechanism**:
    - The core innovation of transformers is the self-attention mechanism, which allows the model to weigh the importance of different words in a sentence relative to each other. This is crucial for understanding context and relationships in language. See [Attention mechanism].
    - **Scaled Dot-Product Attention**: For each word in a sentence, the model computes attention scores with every other word. These scores are used to create a weighted representation of the input, emphasizing relevant words and de-emphasizing less relevant ones.
3. Multi-head attention
    - Instead of having a single attention mechanism, transformers use multiple attention heads. Each head learns different aspects of the relationships between words, allowing the model to capture various linguistic features.
4. **Positional Encoding**:
    - Since transformers do not inherently understand the order of words (unlike Recurrent Neural Networks|RNNs), they use positional encoding to inject information about the position of each word in the sequence.
5. **Feed-Forward Neural Network**:

    - After the attention mechanism, the output is passed through a feed-forward neural network, which is applied independently to each position.
6. **Layer Normalization and Residual Connections**:
    - Transformers use layer normalization and residual connections to stabilize training and help with gradient flow, making it easier to train deep networks.
7. **Training and Applications**:
    - Transformers are trained on large corpora of text data using Unsupervised Learning|unsupervised or semi-supervised learning techniques. They are used for a variety of NLP tasks, including translation, summarization, and question answering.

Additional Concepts

- **Encoder-Decoder Structure**:
    - The encoder processes the input sequence to build a representation, while the decoder takes this representation and generates the output sequence. This setup is particularly useful for tasks like translation.
- **Parallelization**:
    - Unlike Recurrent Neural Networks (Recurrent Neural Networks), transformers do **not require sequential processing,** making them more efficient, especially when training large datasets.

Follow-up questions:

- Transformers vs RNNs