Natural Language Processing

Natural language processing (NLP) is a subfield of artificial intelligence that allows computers to understand, process, and manipulate human language.

Brief History of NLP and Lecture Outline

NLP Evolution Timeline

1950s-1970s: Rule-Based Era

- Turing Test (1950), formal grammars, hand-crafted rules
- Systems like ELIZA (1966) - pattern matching “Rogerian Therapy”chatbot
- Limitations: brittle, didn’t scale, couldn’t handle ambiguity

1980s-2000s: Statistical Revolution

- Shift from rules to learning from data
- N-grams, Hidden Markov Models, machine learning

2006-2017: Deep Learning

- Neural networks applied to NLP
- Word2Vec (2013), RNNs, LSTMs for sequence modeling
- Attention mechanism (2014) - precursor to Transformers

2017-Present: Transformer Era

- Attention is All You Need (2017) - Transformer architecture
- BERT (2018), GPT-2/3 (2019/2020), ChatGPT (2022)
- LLMs with billions of parameters, human-level performance

Lecture Outline

For the rest of this lecture we will cover:

- Text preprocessing and tokenization
- Numerical representations of words
- Language models overview
- NLP Packages Overview
- Named Entity Recognition (NER) with spaCy
- Topic Modeling with BERTopic

Numerical Representation of Words

Numerical Representations of Words

- Machine learning models for NLP are not able to process text in the form of characters and strings.
- Characters and strings must be converted to numbers in order to train our language models.

There are a number of ways to do this. These include

- sparse representations, like one-hot encodings and TF-IDF encodings
- word embeddings.

However, prior to creating a numerical representation of text, we need to **tokenize** the text.

Tokenization

- Tokenization is the process of splitting raw text into smaller pieces, called *tokens*.
- Tokens can be individual characters, words, subwords, or sentences.

Examples of character and word tokenization:

`Show me the money`

Character tokenization:

`['S', 'h', 'o', 'w', 'm', 'e', 't', 'h', 'e', 'm', 'o', 'n', 'e', 'y']`.

Word tokenization:

`['Show', 'me', 'the', 'money']`

Simple Tokenization in Python

Simple python implementation:

```python
# Character and word tokenization

sentence = "Show me the money"
word_tokens = sentence.split()
print(word_tokens)

character_tokens = [char for char in sentence if char != ' ']
print(character_tokens)
```

```
['Show', 'me', 'the', 'money']
['S', 'h', 'o', 'w', 'm', 'e', 't', 'h', 'e', 'm', 'o', 'n', 'e', 'y']
```

Tokenization Methods

- There are advantages and disadvantages to different tokenization methods.
- We showed two very simple strategies.

However, there are other strategies, such as subword and sentence tokenization, see for example:

- Byte-Pair Encoding,
- SentencePiece.

See Andrej Karpathy's Let's build a GPT tokenizer video for a deep dive.

- With tokenization, our goal is to not lose meaning with the tokens. With character based tokenization, especially for English (non-character based languages) we certainly lose meaning.

Huggingface Tokenization

Here is a demo of how to tokenize using the transformers package from Huggingface.

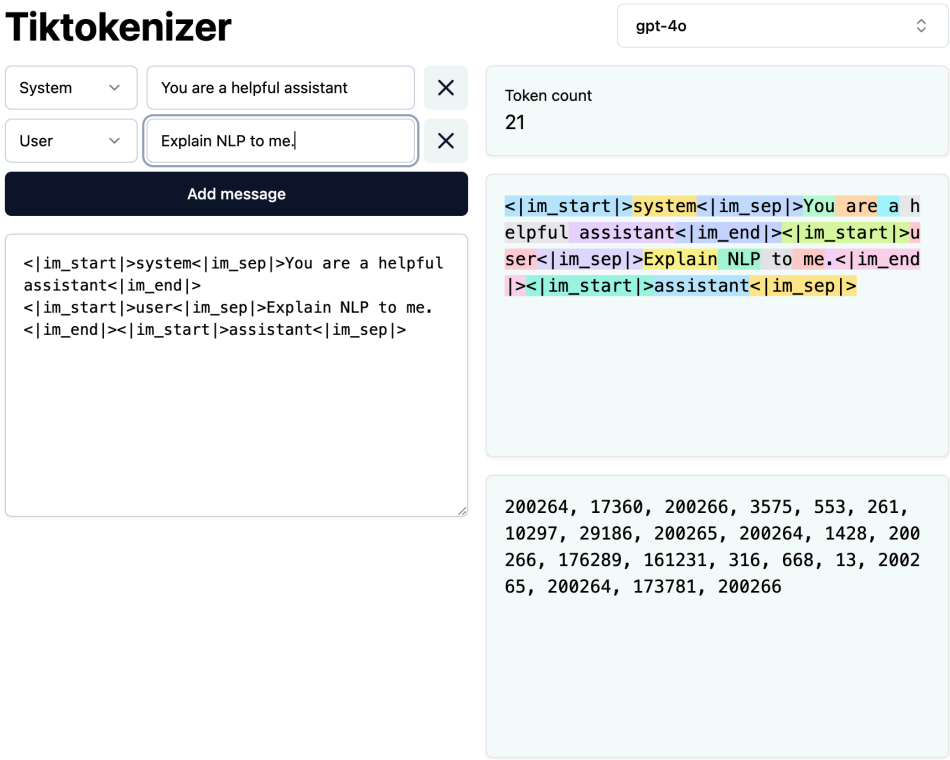
```
from transformers import AutoTokenizer, logging

logging.set_verbosity_warning()

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
tokens = tokenizer.tokenize(sentence)
print(tokens)

# Try a more advanced sentence
sentence2 = "Let's try to see if we can get this transformer to tokenize."
tokens2 = tokenizer.tokenize(sentence2)
print(tokens2)
```

```
['Show', 'me', 'the', 'money']
['Let', "'", 's', 'try', 'to', 'see', 'if', 'we', 'can', 'get', 'this', 'transform', '##er', 'to', 'token',
'##ize', '.']
```

Tokens, Token IDs, and Vocabulary

- Associated to each token is a unique token ID.
- The total number of unique tokens that a model can recognize and process is the *vocabulary size*.
  - The *vocabulary* is the collection of all the unique tokens.
- The tokens (and token ids) alone hold no (semantic) information. What is needed is a numerical representation that *encodes* this information.
- There are different ways to achieve this:
  - One encoding technique that we already considered is one-hot encodings.
  - Another more powerful encoding method, is the creation of word embeddings.

tiktokenizer



tiktokenizer

Tiktokenizer demo

Sparse Representations

We have previously considered the following sparse representations of textual data.

One-Hot Encoding

- Each word is represented as a vector of zeros and a single one.
- Simple but inefficient for large vocabularies.

Example

Given the words cat, dog, and emu here are sample one-hot encodings

$$\begin{aligned} \text{cat} &= [1, 0, 0]^T, \\ \text{dog} &= [0, 1, 0]^T, \\ \text{emu} &= [0, 0, 1]^T. \end{aligned}$$

Bag of Words (BoW)

- Represents text as a collection of word counts.
- Ignores grammar and word order.

Example

Suppose we have the following sentences

1. The cat sat on the mat.
2. The dog sat on the log.
3. The emu sat on the mat.

Sentence	the	cat	sat	on	mat	dog	log	emu
“The cat sat on the mat.”	2	1	1	1	1	0	0	0
“The dog sat on the log.”	2	0	1	1	0	1	1	0
“The emu sat on the mat.”	2	0	1	1	1	0	0	1

TF-IDF (Term Frequency-Inverse Document Frequency)

- Adjusts word frequency by its importance across documents.
- Highlights unique words in a document.
- See Clustering in Practice for more details.

Example

The TF-IDF representations corresponding to the previous sentences.

	cat	dog	log	mat	on	emu	sat	the
Sentence 1	0.4698	0.0000	0.0000	0.4698	0.3546	0.0000	0.3546	0.7093
Sentence 2	0.0000	0.4698	0.4698	0.0000	0.3546	0.0000	0.3546	0.7093
Sentence 3	0.0000	0.0000	0.0000	0.4698	0.3546	0.4698	0.3546	0.7093

Word Embeddings

Word embeddings represent words as **dense vectors in high-dimensional spaces**.

The individual values of the vector may be difficult to interpret, but the overall pattern is that *words with similar meanings are close to each other*, in the sense that their vectors have small angles with each other.

▶ **Question:** Is that true with one-hot encodings?

The similarity of two word embeddings is the cosine of the angle between the two vectors. Recall that for two vectors $v_1, v_2 \in \mathbb{R}^n$, the formula for the cosine of the angle between them is

$$\cos(\theta) = \frac{v_1 \cdot v_2}{\|v_1\|_2 \|v_2\|_2}.$$

Static vs Contextual Embeddings

Word embeddings can be **static** or **contextual**.

Static:
A static embedding is when each word has a single embedding, e.g., Word2Vec.

Contextual:
A contextual embedding (used by more complex language model embedding algorithms) allows the embedding for a word to change depending on its context in a sentence.

Word2Vec: Static Embeddings

Word2Vec (Mikolov et al. 2013) is a technique to learn static word embeddings from large text corpora.

Key characteristics:

- Each word has a **single fixed vector** representation
- Trained to predict context (CBOW) or predict word from context (Skip-gram)
- Captures semantic relationships: *king - man + woman ≈ queen*
- Popular pre-trained models: Google News (300d), GloVe

Advantages:

- Fast to train and use
- Efficient for large vocabularies
- Good for similarity tasks, clustering
- Pre-trained embeddings available

Disadvantages:

- Limited context and no word order information.
- For phrases and sentences, the embedding is the average of the word embeddings.

Contextual vs Static Embeddings

Feature	Static (Word2Vec, GloVe)	Contextual (BERT, GPT)
Representation	One vector per word	Different vectors per context
Example	“bank” always same vector	“bank” differs in “river bank” vs “bank account”
Model Type	Shallow neural network	Deep transformer model
Training	Fast, lightweight	Slow, resource-intensive
Best For	Similarity, clustering, simple tasks	Complex understanding, ambiguity resolution
Dimensionality	100-300 dimensions	768-1024+ dimensions

When to use Word2Vec:

- Limited computational resources
- Task doesn’t require deep context understanding

When to use Contextual Embeddings:

- Need to handle polysemy (words with multiple meanings)
- Complex NLP tasks (NER, question

Language Models

Language Models

A language model is a statistical tool that predicts the probability of a sequence of words. It helps in understanding and generating human language by learning patterns and structures from large text corpora.

1. **N-gram Models**:
    - Predict the next word based on the previous $n-1$ words.
    - Simple and effective for many tasks but limited by fixed context size.
2. **Neural Language Models**:
    - Use neural networks to capture more complex patterns.
    - Examples include *RNNs*, *LSTMs*, and **Transformers**.

See RNNs and LSTMs for more details.

We'll skip N-grams and focus on Transformers.

Transformers

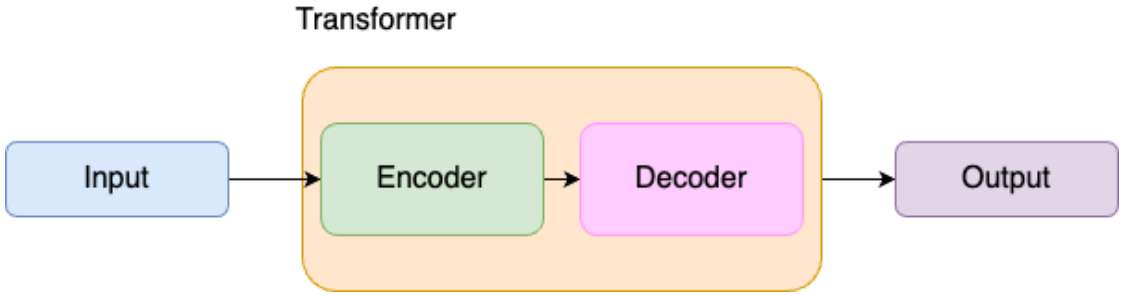
Transformers: High-Level Overview

Transformers (Vaswani et al. 2017) are the foundation of modern NLP systems.

Key innovations:

- Use **attention mechanism** to process entire sequences (e.g. context window) in parallel
  - Although optimizations exist, given huge context window sizes, e.g. Flash Attention (chunking), sparse attention.
- Revolutionized NLP and enabled LLMs (ChatGPT, BERT, GPT-4)
- Scalable across GPUs for massive models

Transformer Architecture

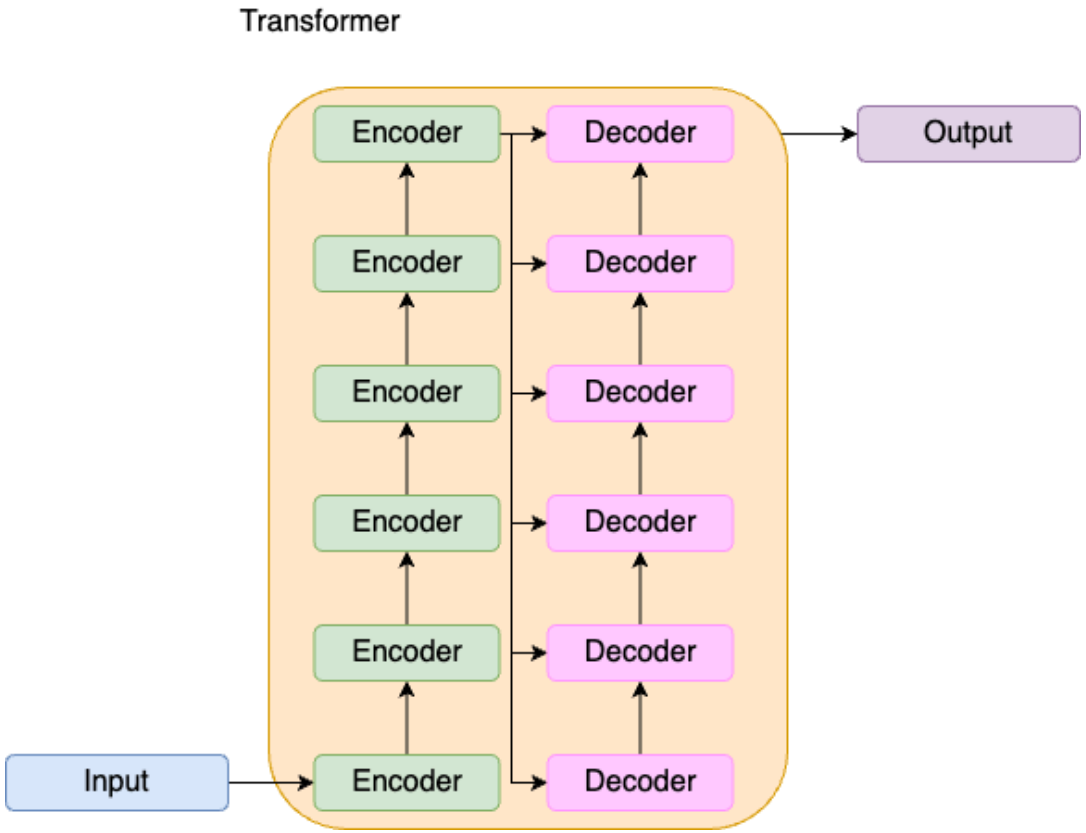


Components:

- **Encoder**: Processes input text, creates rich representations
- **Decoder**: Generates output text using encoder representations
- **Attention**: Allows model to focus on relevant parts of input

> Other variants of the transformer architecture include the encoder-only and decoder-only architectures.

Encoder-Decoder Building Blocks

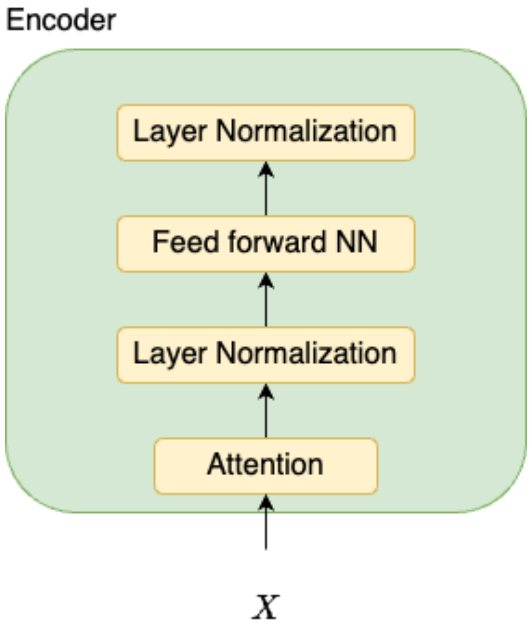


> Can grow the model size by increasing the number of layers and the number of attention heads.

Transformer Building Block

Components:

- Attention mechanism – gathers information from neighboring tokens in the input sequence.
- Feed-forward neural network – processes the information gathered by the attention mechanism.
- Layer normalization – normalizes the output of the feed-forward neural network.



The Attention Mechanism

Attention allows models to understand which words are most relevant to each other.

Example sentence:

> "The elephant didn't cross the river because **it** was tired."

Question: What does "it" refer to?

- Attention helps the model determine that **"it" = "elephant"** (not "river")
- The model "attends" more strongly to "elephant" when processing "it"

How Attention Works (Simplified)

For each word in a sentence:

1. **Calculate relevance scores** with all other words
2. **Apply** softmax to get attention weights (sum to 1)
3. **Compute weighted sum** of word representations
4. Result: each word's representation includes context from relevant words

Multi-head attention: Run multiple attention operations in parallel to capture different types of relationships

> It's a form of adaptive network where some weights are derived from inputs.

Queries, Keys, and Values

The attention mechanism uses three types of vectors for each word:

- **Query (Q)**: “What am I looking for?”
- **Key (K)**: “What do I contain?”
- **Value (V)**: “What information do I have?”

Attention formula:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

- Compute similarity between queries and keys
- Normalize with softmax
- Weight the values by attention scores

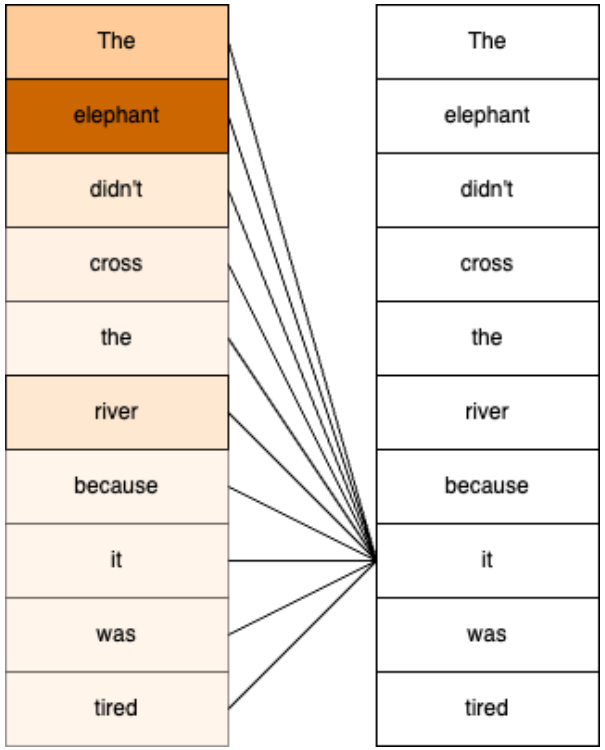
Attention Visualized

Darker connections show stronger attention.

When processing “it”, the model attends most to “elephant”.

This is how transformers handle long-range dependencies and ambiguity.

See also BertViz for a visual exploration of the attention mechanism.



Transformer Architectures

3 Types of Transformer Models

1. **Encoder-Decoder** – Sequence-to-sequence tasks (e.g., machine translation)
    - Example: Original Transformer, T5
2. **Encoder-Only** – Transforms text embeddings into latent representations for understanding and classification tasks
    - Example: **BERT** (sentiment analysis, NER, classification)
3. **Decoder-Only** – Predicts next token used for text generation and completion
    - Example: **GPT** (ChatGPT, text generation, AI assistants)

BERT: Encoder Model

BERT (Bidirectional Encoder Representations from Transformers) (Devlin et al. 2019)

Key features:

- Reads text **bidirectionally** (considers context from both directions)
- Pre-trained on masked language modeling (predict hidden words)
- 340M parameters (BERT-Large), 110M (BERT-Base)
- Fine-tuned for specific tasks with one additional layer

Common uses: Text classification, NER, question answering, sentiment analysis

GPT: Decoder Model

GPT (Generative Pre-trained Transformer) (Brown et al. 2020)

Key features:

- Reads text **left-to-right** (autoregressive)
- Predicts next token: $P(t_1, t_2, \ldots, t_N) = P(t_1) \prod_{n=2}^{N} P(t_n | t_1, \ldots t_{n-1})$
- GPT-3: 175 billion parameters
- Excellent at text generation

Common uses: Text completion, creative writing, chatbots, code generation

Transformer Applications

Transformers enable powerful NLP applications:

- **Machine Translation**: Translate between languages
- **Text Summarization**: Generate concise summaries
- **Question Answering**: Answer questions from context
- **Named Entity Recognition**: Identify entities in text
- **Sentiment Analysis**: Determine sentiment/emotion
- **Text Generation**: Create human-like text

NLP Packages Overview

NLTK: Natural Language Toolkit

NLTK is one of the most comprehensive Python libraries for NLP, created for teaching and research.

Key Features:

- Extensive text processing tools (tokenization, stemming, tagging, parsing)
- Access to 50+ corpora and lexical resources (WordNet, TreeBank)
- Sentiment analysis, text classification, and chunking

Best For:

- Learning *classical* NLP concepts and fundamentals
- **Academic research and prototyping**
- Working with linguistic data structures
- Access to curated corpora

Limitations:

- Slower than modern libraries
- Less suited for production
- Requires more manual pipeline construction

spaCy: Production-Ready NLP

spaCy is a modern, industrial-strength NLP library designed for production use.

Key Features:

- Fast and efficient (Cython-optimized)
- Pre-trained models for 70+ languages
- Built-in NER, POS tagging, dependency parsing
- Word vectors and document similarity
- Beautiful visualization tools (displaCy)
- Easy integration with deep learning (PyTorch, TensorFlow)

Best For:

- Production NLP pipelines
- Real-time text processing
- Information extraction at scale
- Named Entity Recognition
- Document analysis and classification

BERTopic: Overview

BERTopic (Grootendorst 2022) is a modern topic modeling technique using transformer embeddings.

More later…

LangChain: Overview

LangChain (**langchain2023?**) is a framework for building applications that use language models.

Key advantages:

- Easy to use
- Built-in support for many language models
- Built-in support for many NLP tasks

Common uses:

- Building chatbots and virtual assistants
- Building information extraction systems
- Building summarization systems, …

NLP Applications

We'll dive a bit deeper into two of the most common NLP applications:

- Named entity recognition
- Topic modeling

Named Entity Recognition (NER)

What is Named Entity Recognition?

Named Entity Recognition (NER) is the task of identifying and classifying named entities in text into predefined categories.

Common entity types:

- **PERSON**: Names of people (e.g., “Barack Obama”)
- **ORGANIZATION**: Companies, agencies (e.g., “Google”, “FBI”)
- **LOCATION**: Cities, countries, landmarks (e.g., “Paris”, “Mount Everest”)
- **DATE**: Dates and times (e.g., “January 1, 2024”)
- **MONEY**: Monetary values (e.g., “$100”)
- **GPE**: Geopolitical entities (countries, cities, states)

Applications: Information extraction, content classification, question answering, knowledge graphs

NER with spaCy: Setup

You need to download the model you want to use.

Installation:

```
pip install spacy
python -m spacy download en_core_web_sm  # Small model
```

Available models:

- `en_core_web_sm`: Small (12 MB) - fast, good accuracy
- `en_core_web_md`: Medium (40 MB) - word vectors included
- `en_core_web_lg`: Large (560 MB) - best accuracy, full vectors

spaCy NER: Basic Usage

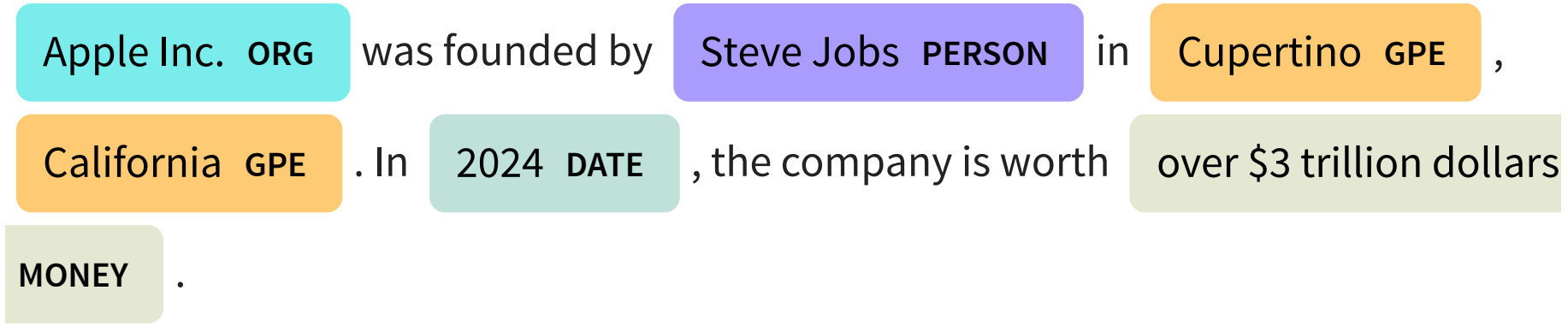
```python
import spacy

# Load pre-trained model
nlp = spacy.load("en_core_web_sm")

text = "Apple Inc. was founded by Steve Jobs in Cupertino, California. In 2024, the company is worth over $3

# Process text - creates Doc object
doc = nlp(text)

# Extract named entities
print("Entities found:")
for ent in doc.ents:
    print(f"  {ent.text:20} -> {ent.label_:15} ({spacy.explain(ent.label_)})")
```

```
Entities found:
  Apple Inc.           -> ORG             (Companies, agencies, institutions, etc.)
  Steve Jobs           -> PERSON          (People, including fictional)
  Cupertino            -> GPE             (Countries, cities, states)
  California           -> GPE             (Countries, cities, states)
  2024                 -> DATE            (Absolute or relative dates or periods)
  over $3 trillion dollars -> MONEY           (Monetary values, including unit)
```

spaCy NER: Visualization

Built-in visualization with `displacy`:

```
from spacy import displacy

# Visualize entities inline
displacy.render(doc, style="ent", jupyter=True)
```



Key features:

- Highlights entities with color coding
- Shows entity labels on hover
- Can export as HTML or SVG
- Customizable styles

spaCy NER: Advanced Features

```python
# Access entity properties
for ent in doc.ents:
    print(f"{ent.text:20} | Label: {ent.label_:10} | Start: {ent.start_char:3} | End: {ent.end_char:3}")

# Filter by entity type
orgs = [ent.text for ent in doc.ents if ent.label_ == "ORG"]
print(f"\nOrganizations: {orgs}")

# Entity spans and context
for ent in doc.ents:
    print(f"{ent.text} → Sentence: {ent.sent}")
```

```
Apple Inc.           | Label: ORG        | Start:   0 | End:  10
Steve Jobs           | Label: PERSON     | Start:  26 | End:  36
Cupertino            | Label: GPE        | Start:  40 | End:  49
California           | Label: GPE        | Start:  51 | End:  61
2024                 | Label: DATE       | Start:  66 | End:  70
over $3 trillion dollars | Label: MONEY      | Start:  93 | End: 117

Organizations: ['Apple Inc.']
Apple Inc. → Sentence: Apple Inc. was founded by Steve Jobs in Cupertino, California.
Steve Jobs → Sentence: Apple Inc. was founded by Steve Jobs in Cupertino, California.
Cupertino → Sentence: Apple Inc. was founded by Steve Jobs in Cupertino, California.
California → Sentence: Apple Inc. was founded by Steve Jobs in Cupertino, California.
2024 → Sentence: In 2024, the company is worth over $3 trillion dollars.
over $3 trillion dollars → Sentence: In 2024, the company is worth over $3 trillion dollars.
```

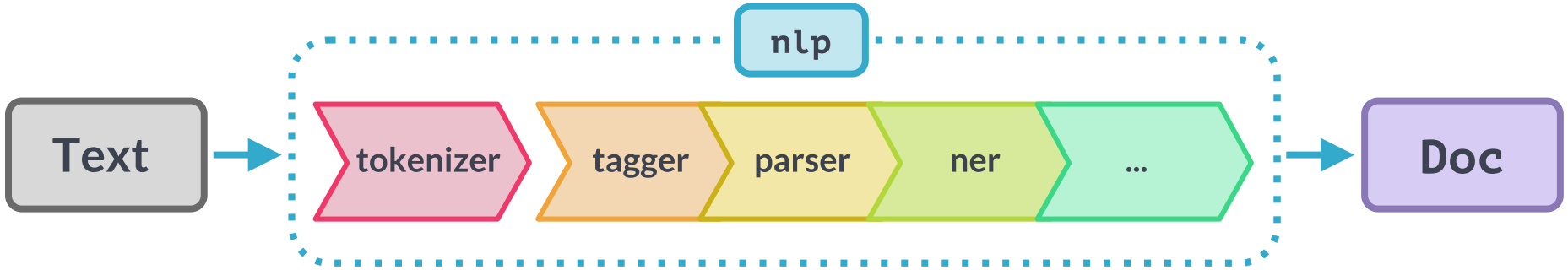
spaCy Pipeline Architecture

```
# View pipeline components
print("Pipeline components:")
for name, component in nlp.pipeline:
    print(f"  - {name}")
```

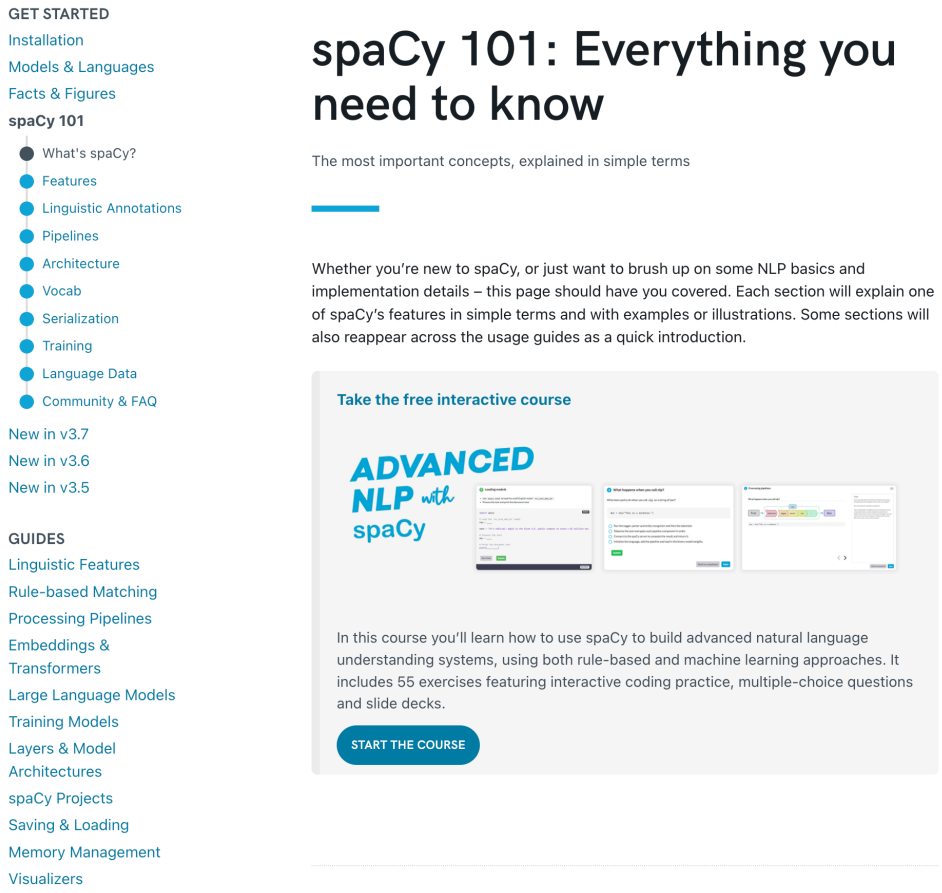
```
Pipeline components:
  - tok2vec
  - tagger
  - parser
  - attribute_ruler
  - lemmatizer
  - ner
```

Processing steps:

1. **Tokenizer**: Split text into tokens
2. **tok2vec**: Convert tokens to vectors
3. **Tagger**: Part-of-speech tagging
4. **Parser**: Dependency parsing
5. **NER**: Named entity recognition
6. **Lemmatizer**: Word lemmatization



spaCy – Dive Deeper



https://spacy.io/usage/spacy-101

Topic Modeling

What is Topic Modeling?

Topic modeling is an unsupervised learning technique that discovers abstract "topics" in a collection of documents.

Key concepts:

- **Topic**: A distribution over words (e.g., "sports" → {game, team, player, score, …})
- **Document**: A mixture of topics
- **Goal**: Automatically discover topics and their distributions

Applications:

- Content recommendation and discovery
- Document organization and clustering
- Trend analysis over time
- Search and information retrieval
- Content summarization

Classical approaches (LDA, NMF) use bag-of-words representations and require specifying the number of topics.

Modern approaches (BERTopic) leverage transformer embeddings and can automatically determine topics.

BERTopic: Overview

BERTopic (Grootendorst 2022) is a modern topic modeling technique using transformer embeddings.

Key advantages:

- Uses **semantic embeddings** from BERT (captures context and meaning)
- **Automatically determines** optimal number of topics
- Excellent for **short documents** (tweets, reviews, titles)
- Produces **highly coherent** and interpretable topics
- Built-in **interactive visualizations**

When to use BERTopic:

- Need high-quality, coherent topics
- Working with social media, news articles, reviews
- Want to avoid hyperparameter tuning (number of topics)
- Have GPU resources available
- Need to track topics over time

BERTopic: How It Works

BERTopic uses a modular pipeline with four main steps:

1. Document Embeddings

- Use pre-trained BERT/transformer models
- Each document → 768-dimensional vector (or higher)
- Captures semantic meaning and context

2. Dimensionality Reduction (UMAP)

- Reduce from 768 dimensions to ~5 dimensions
- Preserves local and global structure
- Enables efficient clustering

3. Clustering (HDBSCAN)

- Density-based clustering algorithm
- Automatically determines number of clusters
- Assigns outliers to -1 topic

4. Topic Representation (c-TF-IDF)

- Class-based TF-IDF weights words by topic
- Extracts most representative words per topic
- Creates interpretable topic labels

BERTopic: Installation and Setup

Installation:

```
pip install bertopic
pip install umap-learn hdbscan  # clustering dependencies
```

Quick start:

```
from bertopic import BERTopic

# Initialize with default settings
topic_model = BERTopic(language="english", calculate_probabilities=True)

# Fit and transform
topics, probs = topic_model.fit_transform(documents)
```

BERTopic: Basic Example

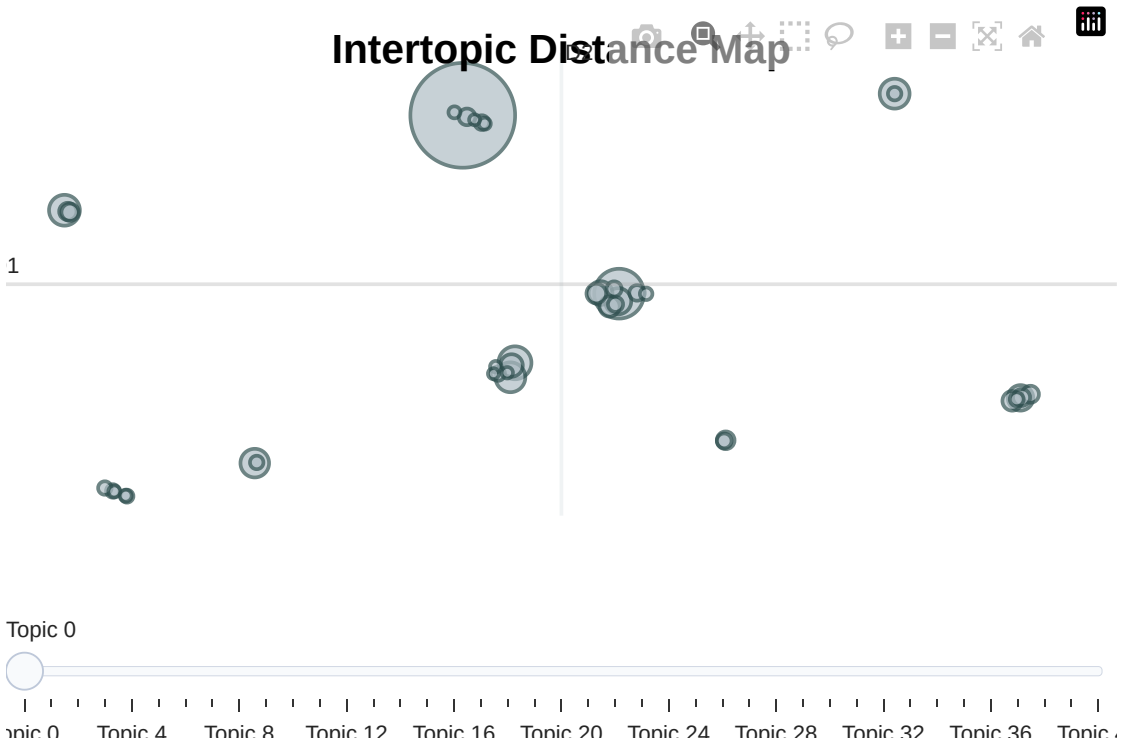
```python
from bertopic import BERTopic
from sklearn.datasets import fetch_20newsgroups

categories = ['sci.space', 'rec.sport.baseball', 'comp.graphics']
docs = fetch_20newsgroups(subset='all', categories=categories, remove=('headers', 'footers', 'quotes'))['data
topic_model = BERTopic()

topics, probs = topic_model.fit_transform(docs)

print(f"Number of topics found: {len(set(topics)) - (1 if -1 in topics else 0)}")
print(f"Outlier documents: {sum(1 for t in topics if t == -1)}")
```

Loading weights: 100% 103/103 [00:00<00:00, 6250.46it/s]

```
Number of topics found: 41
Outlier documents: 702
```

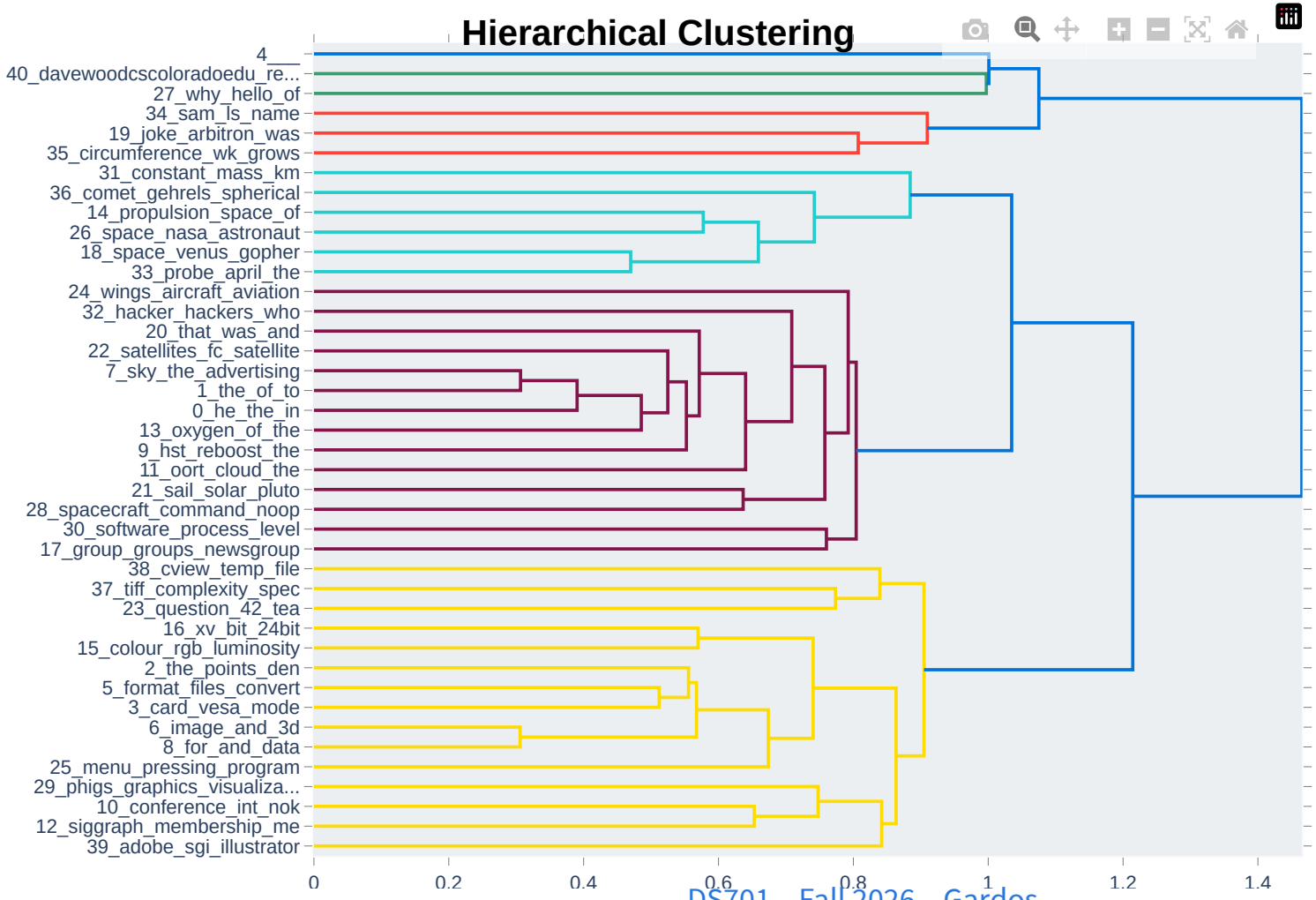
BERTopic: Visualize Topics

▶ Code



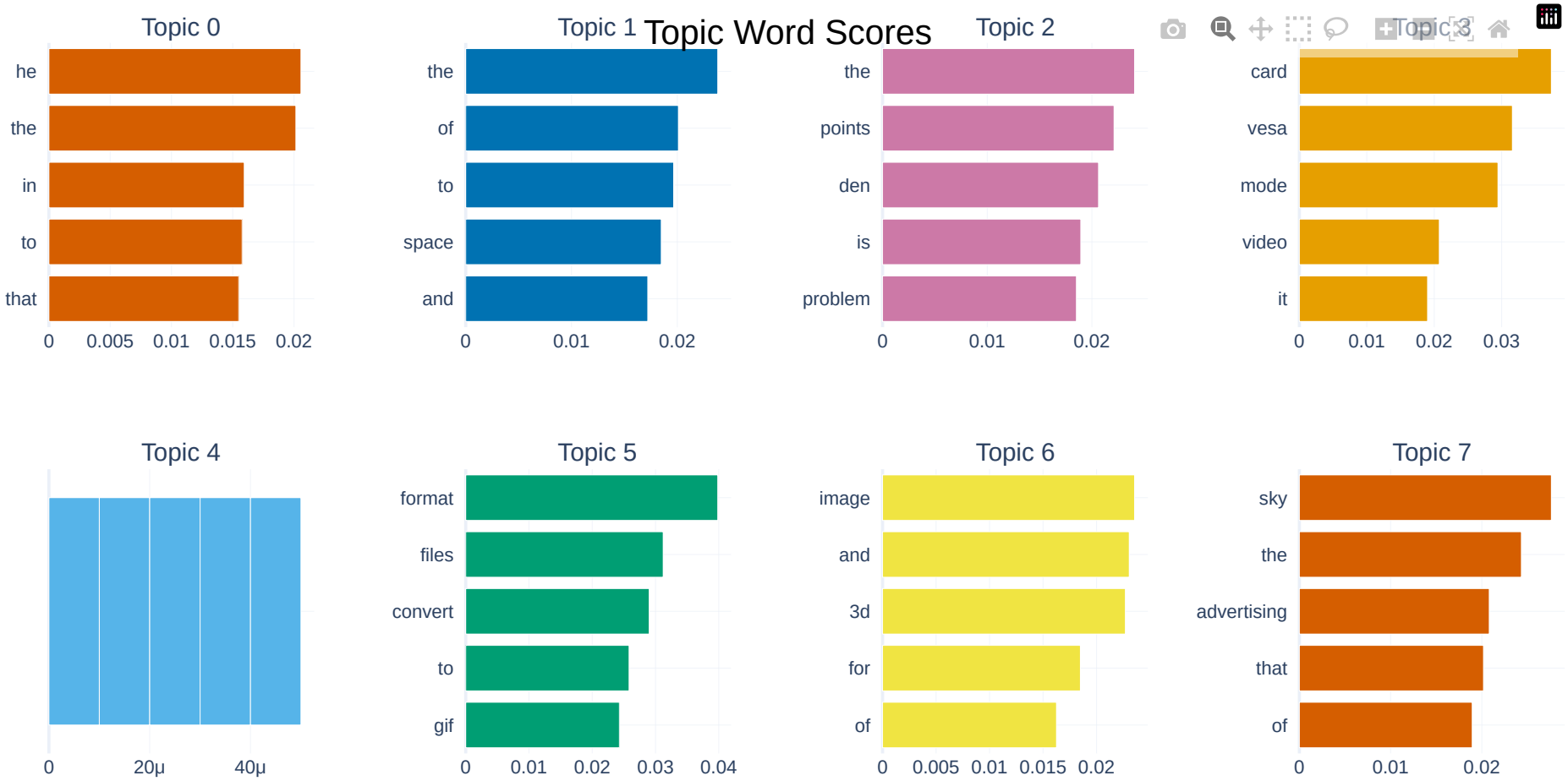
BERTopic: Visualize Topic Hierarchy

▶ Code



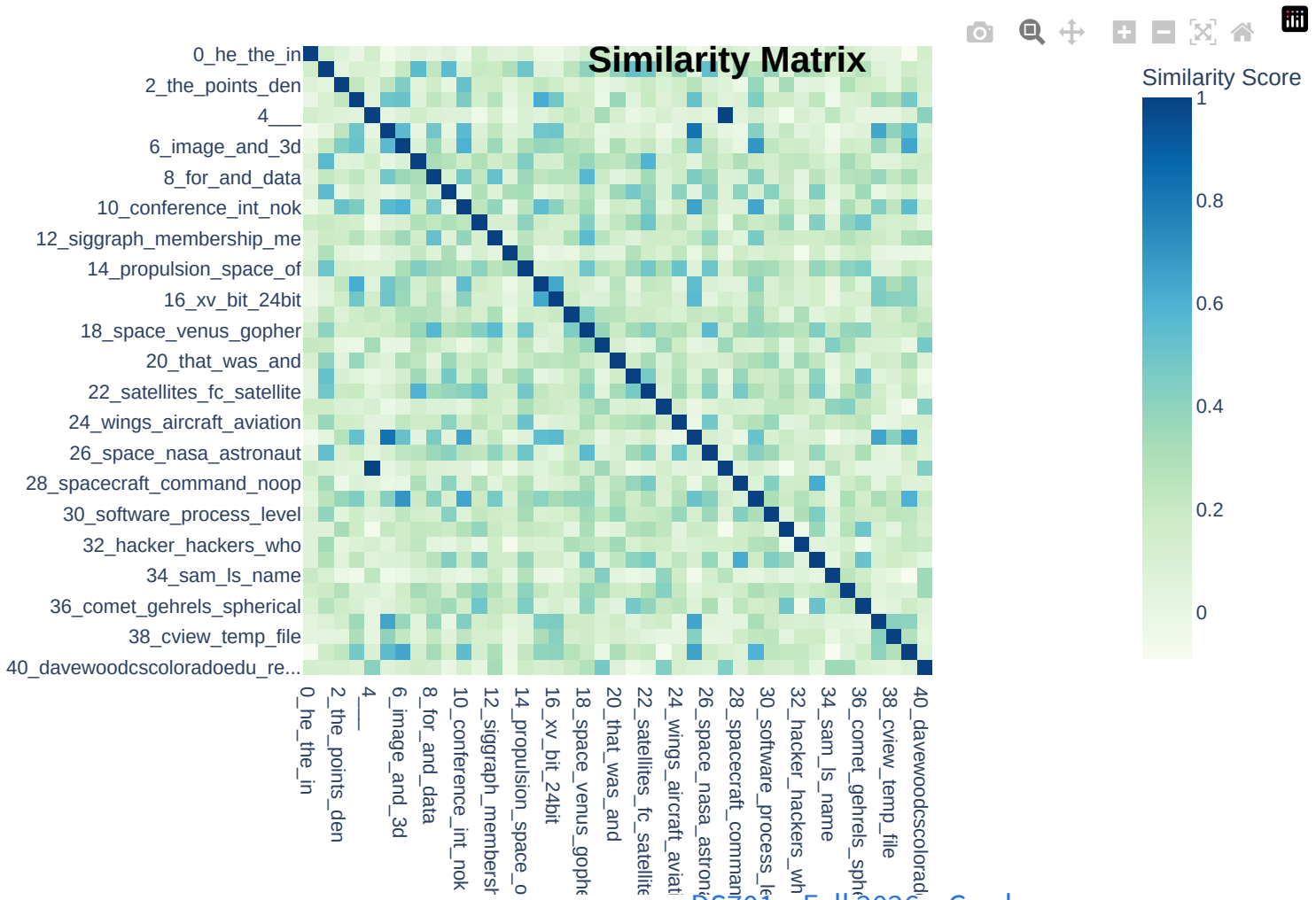
BERTopic: Visualize Topic Bar Chart

▶ Code



BERTopic: Visualize Topic Heatmap

▶ Code



BERTopic: Find Similar Topics

```
similar_topics, similarity = topic_model.find_topics("space exploration and satellites", top_n=3)
print(f"Topics similar to 'space exploration and satellites': {similar_topics}")
```

```
Topics similar to 'space exploration and satellites': [22, 26, 1]
```

BERTopic: Exploring Topics

```python
# Get topic information
topic_info = topic_model.get_topic_info()
print("Topic Overview:")
print(topic_info[['Topic', 'Count', 'Name']].head(10))

# Get representative documents for a topic
topic_id = 0
rep_docs = topic_model.get_representative_docs(topic_id)
print(f"\nRepresentative documents for Topic {topic_id}:")
for i, doc in enumerate(rep_docs[:2], 1):
    print(f"{i}. {doc[:100]}...")
```

```
Topic Overview:
   Topic  Count                        Name
0     -1    702           -1_the_of_to_and
1      0    912             0_he_the_in_to
2      1    206          1_the_of_to_space
3      2     90        2_the_points_den_is
4      3     79     3_card_vesa_mode_video
5      4     74                    4____
6      5     74   5_format_files_convert_to
7      6     68         6_image_and_3d_for
8      7     57  7_sky_the_advertising_that
9      8     54         8_for_and_data_ftp

Representative documents for Topic 0:
1.
    Oh, yeah.  Dave Winfield--marginal player.  Guy didn't hit a lick, had
negligible power, was a cra...
2.
I am trying to think how to respond to this without involving personal feeling
```

BERTopic: Topic Words

```python
# Show top words for each topic
print("Top words per topic:\n")
for topic_num in sorted(set(topics)):
    if topic_num == -1:  # Skip outlier topic
        continue
    words = topic_model.get_topic(topic_num)
    if words:
        # Get word and score
        top_words = ', '.join([f"{word}({score:.2f})" for word, score in words[:8]])
        print(f"Topic {topic_num}: {top_words}")
```

```
Top words per topic:

Topic 0: he(0.02), the(0.02), in(0.02), to(0.02), that(0.02), and(0.01), his(0.01), was(0.01)
Topic 1: the(0.02), of(0.02), to(0.02), space(0.02), and(0.02), that(0.01), in(0.01), is(0.01)
Topic 2: the(0.02), points(0.02), den(0.02), is(0.02), problem(0.02), of(0.02), this(0.02), algorithm(0.02)
Topic 3: card(0.04), vesa(0.03), mode(0.03), video(0.02), it(0.02), driver(0.02), vga(0.02), to(0.02)
Topic 4: (0.00), (0.00), (0.00), (0.00), (0.00), (0.00), (0.00), (0.00)
Topic 5: format(0.04), files(0.03), convert(0.03), to(0.03), gif(0.02), file(0.02), me(0.02), it(0.02)
Topic 6: image(0.02), and(0.02), 3d(0.02), for(0.02), of(0.02), it(0.02), or(0.02), the(0.01)
Topic 7: sky(0.03), the(0.02), advertising(0.02), that(0.02), of(0.02), to(0.02), rights(0.02), it(0.02)
Topic 8: for(0.02), and(0.02), data(0.02), ftp(0.02), available(0.02), the(0.01), in(0.01), from(0.01)
Topic 9: hst(0.05), reboost(0.03), the(0.03), mission(0.02), shuttle(0.02), to(0.02), mass(0.02), is(0.02)
Topic 10: conference(0.04), int(0.03), nok(0.03), oprows(0.02), opcols(0.02), for(0.02), sas(0.02), on(0.02)
Topic 11: oort(0.04), cloud(0.03), the(0.03), grbs(0.03), distribution(0.03), of(0.03), burst(0.02),
detectors(0.02)
Topic 12: siggraph(0.05), membership(0.03), me(0.03), to(0.02), my(0.02), send(0.02), you(0.02), address(0.02)
Topic 13: oxygen(0.03), of(0.02), the(0.02), to(0.02), in(0.02), is(0.02), pressure(0.02), it(0.02)
Topic 14: propulsion(0.03), space(0.02), of(0.02), and(0.02), the(0.02), fusion(0.02), lunar(0.02), was(0.02)
Topic 15: colour(0.04), rgb(0.03), luminosity(0.03), colours(0.02), color(0.02), bits(0.02), green(0.02),
bit(0.02)
```

BERTopic: Finding Similar Topics

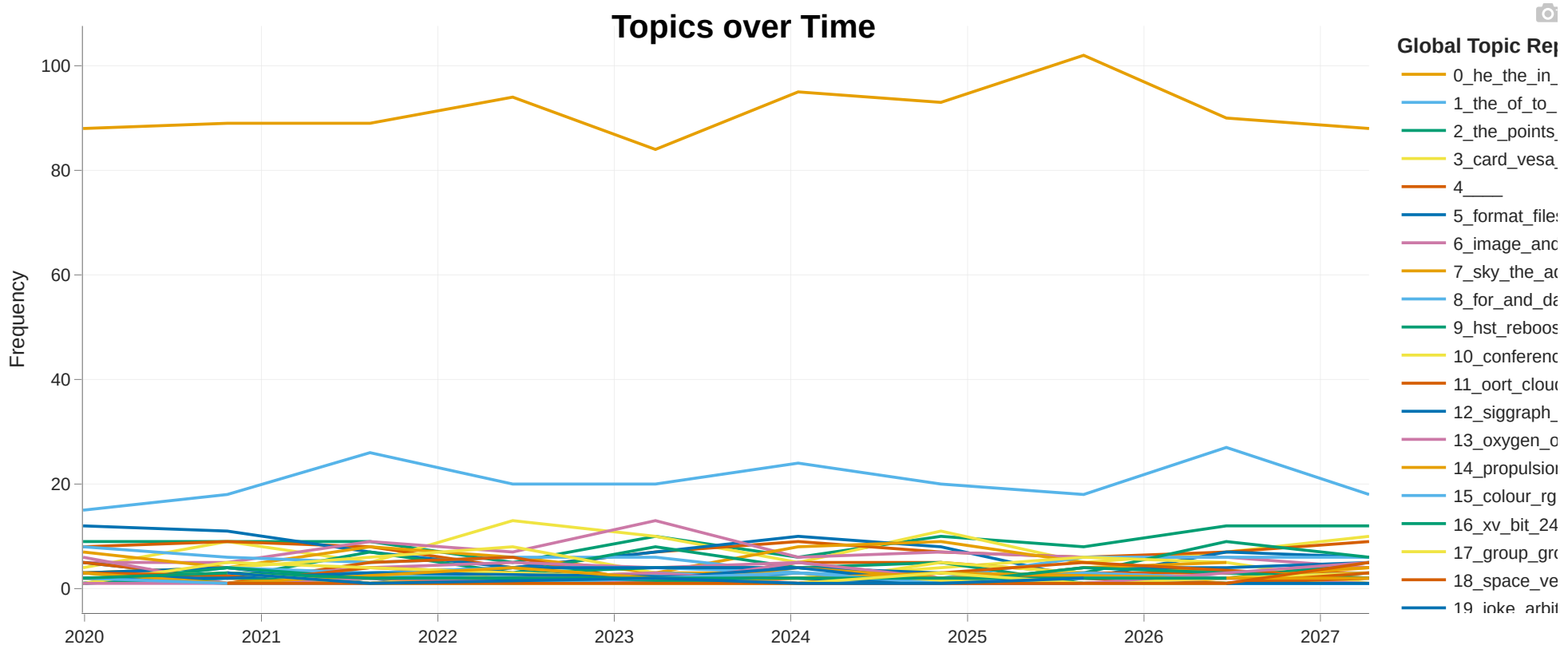
```python
# Find topics similar to a search query
query = "space exploration and satellites"
similar_topics, similarity = topic_model.find_topics(query, top_n=3)

print(f"Topics similar to '{query}':")
for topic_id, score in zip(similar_topics, similarity):
    if topic_id != -1:
        words = topic_model.get_topic(topic_id)
        top_words = ', '.join([word for word, _ in words[:5]])
        print(f"  Topic {topic_id} (similarity: {score:.3f}): {top_words}")
```

```
Topics similar to 'space exploration and satellites':
  Topic 22 (similarity: 0.567): satellites, fc, satellite, the, of
  Topic 26 (similarity: 0.520): space, nasa, astronaut, and, center
  Topic 1 (similarity: 0.502): the, of, to, space, and
```

BERTopic: Topic Over Time

BERTopic can track how topics evolve over time:

▶ Code



Use cases:

BERTopic: Customization

BERTopic is highly customizable:

```
from sentence_transformers import SentenceTransformer
from umap import UMAP
from hdbscan import HDBSCAN

# Custom embedding model
embedding_model = SentenceTransformer("all-MiniLM-L6-v2")

# Custom UMAP settings
umap_model = UMAP(n_neighbors=15, n_components=5, min_dist=0.0)

# Custom HDBSCAN settings
hdbscan_model = HDBSCAN(min_cluster_size=15, min_samples=10)

# Create custom BERTopic model
topic_model = BERTopic(
    embedding_model=embedding_model,
    umap_model=umap_model,
    hdbscan_model=hdbscan_model,
    verbose=False
)
```

BERTopic: Practical Tips

Best practices:

- Use at least 100-200 documents per expected topic
- For short texts, use smaller `min_topic_size` (5-10)
- For long documents, increase `min_topic_size` (20-50)
- Experiment with different embedding models
- Save models for reuse: `topic_model.save("my_model")`

Common issues:

- Too many outliers? Lower `min_cluster_size`
- Topics too broad? Increase `min_topic_size`
- Slow performance? Use smaller embedding model
- Poor topics? Try different embedding model or more documents

Summary

NLP Tools and Applications Recap

What we covered:

1. **Text representation**: Sparse (TF-IDF) vs embeddings (Word2Vec vs BERT)
2. **Language models**: N-grams → Transformers
3. **Transformers**: Attention mechanism (QKV), encoder-decoder architecture
4. **Transformer architectures**: BERT (encoder), GPT (decoder)
5. **NLP Tools**: NLTK (learning) vs spaCy (production)
6. **Named Entity Recognition**: spaCy for production-ready NER
7. **Topic Modeling**: BERTopic with transformer embeddings

References

Brown, Tom B, Benjamin Mann, Nick Ryder, et al. 2020. “Language Models Are Few-Shot Learners.” *arXiv Preprint arXiv:2005.14165*, ahead of print. https://doi.org/10.48550/arXiv.2005.14165.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. “BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding.” *arXiv Preprint arXiv:1810.04805*, ahead of print. https://doi.org/10.48550/arXiv.1810.04805.

Grootendorst, Maarten. 2022. “BERTopic: Neural Topic Modeling with a Class-Based TF-IDF Procedure.” *arXiv Preprint arXiv:2203.05794*. https://arxiv.org/abs/2203.05794.

Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. “Efficient Estimation of Word Representations in Vector Space.” *Proceedings of Workshop at ICLR*. https://arxiv.org/abs/1301.3781.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, et al. 2017. “Attention Is All You Need.” *Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS 2017)* (Long Beach, CA, USA), 11. https://arxiv.org/abs/1706.03762.