Chapter 3. Text Clustering and Topic Modeling

Although supervised techniques, such as classification, have reigned supreme over the last few years in the industry, the potential of unsupervised techniques such as text clustering cannot be understated.

what is text clustering ?

Text clustering aims to group similar texts based on their semantic content, meaning, and relationships, as illustrated in Figure 3-1. Just like how we've used distances between text embeddings in dense retrieval in chapter XXX, clustering embeddings allow us to group the documents in our archive by similarity.

The resulting clusters of semantically similar documents not only facilitate efficient categorization of large volumes of unstructured text but also allows for quick exploratory data analysis. With the advent of Large Language Models (LLMs) allowing for contextual and semantic representations of text, the power of text clustering has grown significantly over the last years. Language is not a bag of words, and Large Language Models have proved to be quite capable of capturing that notion.

An underestimated aspect of text clustering is its potential for creative solutions and implementations. In a way, unsupervised means that we are not constrained by a certain task or thing that we want to optimize. As a result, there is much freedom in text clustering that allows us to steer from the well-trodden paths. Although text clustering would naturally be used for grouping and classifying documents, it can be used to algorithmically and visually find improper labels, perform topic modeling, speed up labeling, and many more interesting use cases.

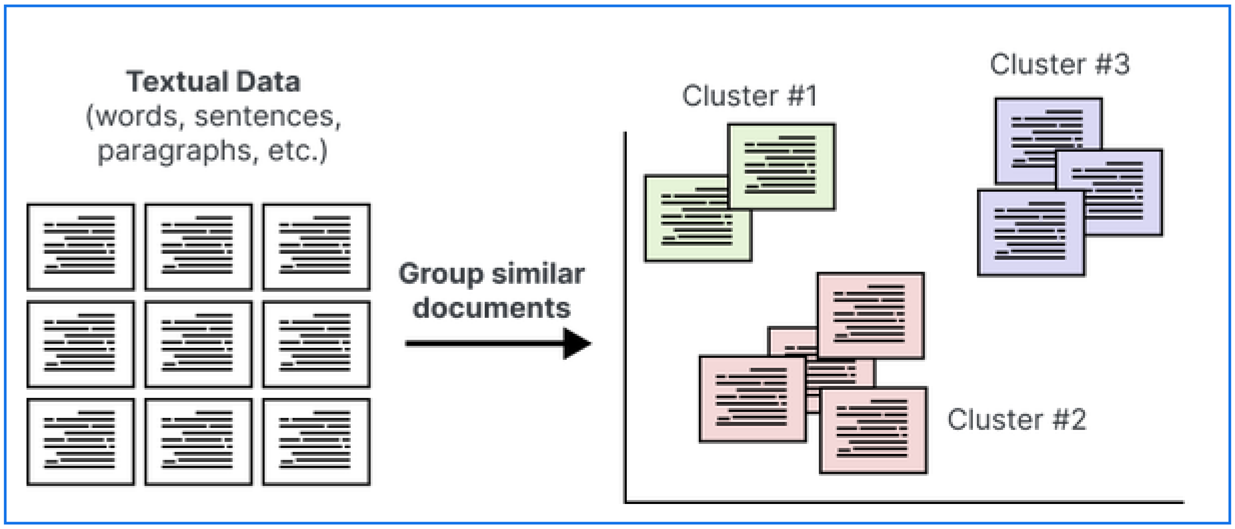


Figure 3-1. Clustering unstructured textual data.

This freedom also comes with its challenges. Since we are not guided by a specific task, then how do we evaluate our unsupervised clustering output? How do we optimize our algorithm? Without labels, what are we optimizing the algorithm for? When do we know our algorithm is correct? What does it mean for the algorithm to be "correct"? Although these challenges can be quite complex, they are not insurmountable but often require some creativity and a good understanding of the use case.

Striking a balance between the freedom of text clustering and the challenges it brings can be quite difficult. This becomes even more pronounced if we step into the world of topic modeling, which has started to adopt the “text clustering” way of thinking.

what is topic modeling?

With topic modeling, we want to discover abstract topics that appear in large collections of textual data. We can describe a topic in many ways, but it has traditionally been described by a set of keywords or key phrases. A topic about natural language processing (NLP) could be described with terms such as “deep learning”, “transformers”, and “self-attention”. Traditionally, we expect a document about a specific topic to contain terms appearing more frequently than others. This expectation, however, ignores contextual information that a document might contain. Instead, we can leverage Large Language Models, together with text clustering, to model contextualized textual information and extract semantically-informed topics. Figure 3-2 demonstrates this idea of describing clusters through textual representations.

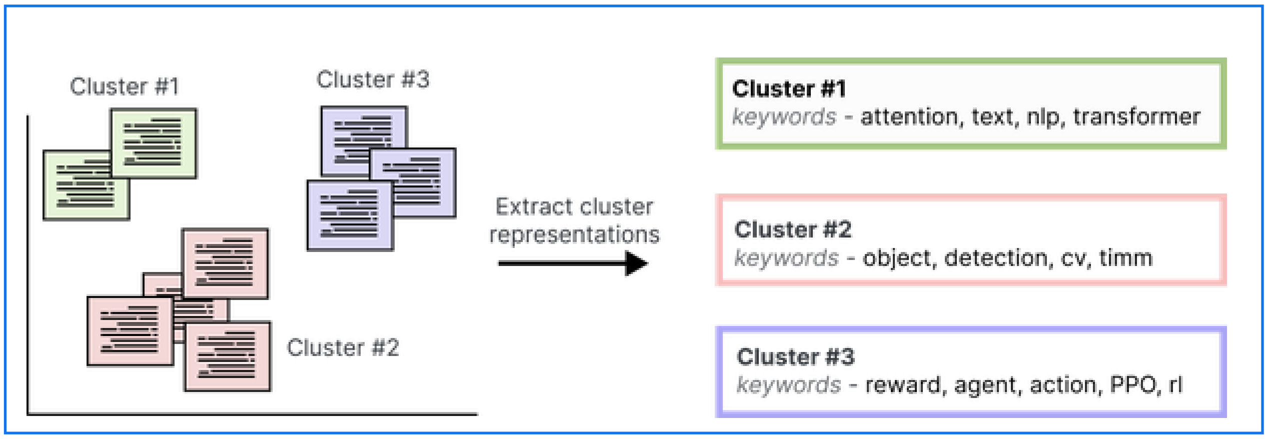

Cluster #1
Cluster #3
Cluster #2
Extract cluster representations
Cluster #1
keywords - attention, text, nlp, transformer
Cluster #2
keywords - object, detection, cv, timm
Cluster #3
keywords - reward, agent, action, PPO, rl

Figure 3-2. Topic modeling is a way to give meaning to clusters of textual documents.

In this chapter, we will provide a guide on how text clustering can be done with Large Language Models. Then, we will transition into a text-clustering-inspired method of topic modeling, namely BERTopic.

Text Clustering

what is text clustering ?

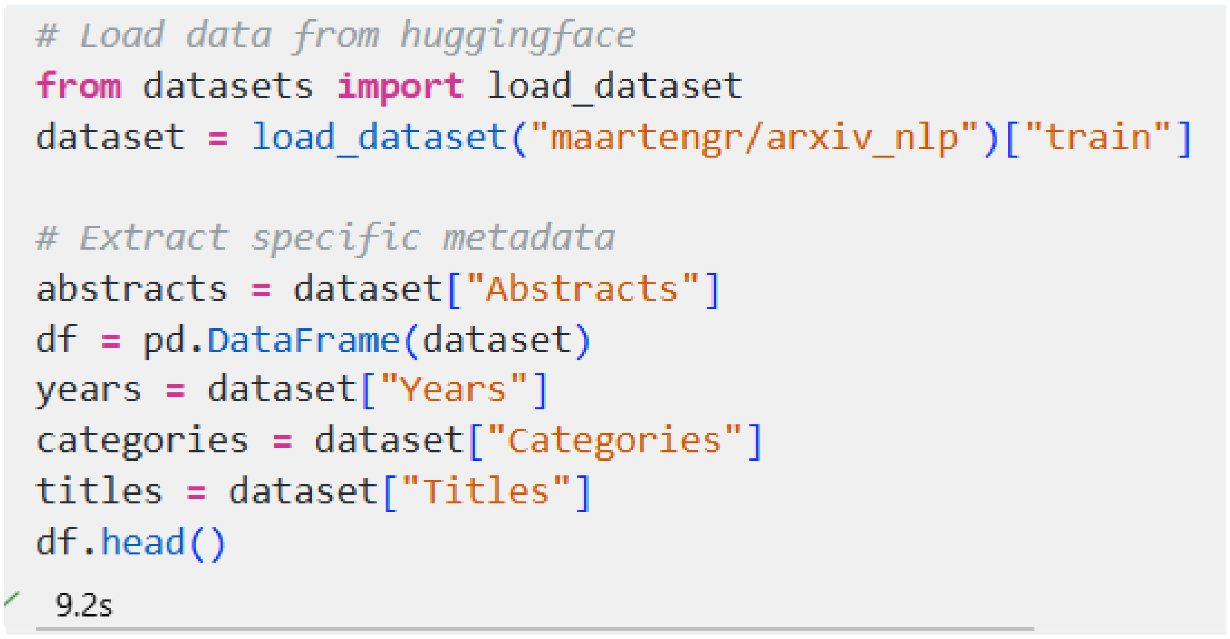
One major component of exploratory data analysis in NLP is text clustering. This unsupervised technique aims to group similar texts or documents together as a way to easily discover patterns among large collections of textual data. Before diving into a classification task, text clustering allows for getting an intuitive understanding of the task but also of its complexity.

The patterns that are discovered from text clustering can be used across a variety of business use cases. From identifying recurring support issues and discovering new content to drive SEO practices, to detecting topic trends in social media and discovering duplicate content. The possibilities are diverse and with such a technique, creativity becomes a key component. As a result, text clustering can become more than just a quick method for exploratory data analysis.

Data

Before we describe how to perform text clustering, we will first introduce the data that we are going to be using throughout this chapter. To keep up with the theme of this book, we will be clustering a variety of ArXiv articles in the domain of machine learning and natural language processing. The dataset contains roughly **XXX** articles between **XXX** and **XXX**.

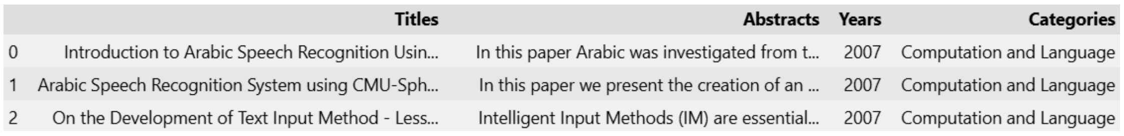
We start by importing our dataset using HuggingFace's dataset package and extracting metadata that we are going to use later on, like the abstracts, years, and categories of the articles.



```python
# Load data from huggingface
from datasets import load_dataset
dataset = load_dataset("maartengr/arxiv_nlp")["train"]

# Extract specific metadata
abstracts = dataset["Abstracts"]
df = pd.DataFrame(dataset)
years = dataset["Years"]
categories = dataset["Categories"]
titles = dataset["Titles"]
df.head()
```

```
 9.2s
```



```
                                         Titles                                          Abstracts  Years               Categories
0     Introduction to Arabic Speech Recognition Usin...   In this paper Arabic was investigated from t...   2007  Computation and Language
1   Arabic Speech Recognition System using CMU-Sph...    In this paper we present the creation of an ...   2007  Computation and Language
2    On the Development of Text Input Method - Less...    Intelligent Input Methods (IM) are essential...   2007  Computation and Language
```

Now that we have our data, we can perform text clustering. To perform text clustering, a number of techniques can be employed, from graph-based neural networks to centroid-based clustering techniques. In this section, we will go through a well-known pipeline for text clustering that consists of three major steps:

how will we do text clustring ?

1. Embed documents
2. Reduce dimensionality
3. Cluster embeddings

1. Embed documents

The first step in clustering textual data is converting our textual data to text embeddings. Recall from previous chapters that embeddings are numerical representations of text that capture its meaning. Producing embeddings optimized for semantic similarity tasks is especially important for clustering. By mapping each document to a numerical representation such that semantically similar documents are close, clustering will become much more powerful. A set of popular Large Language Models optimized for these kinds of tasks can be found in the well-known sentence-transformers framework (reimers2019sentence). Figure 3-3 shows this first step of converting documents to numerical representations.

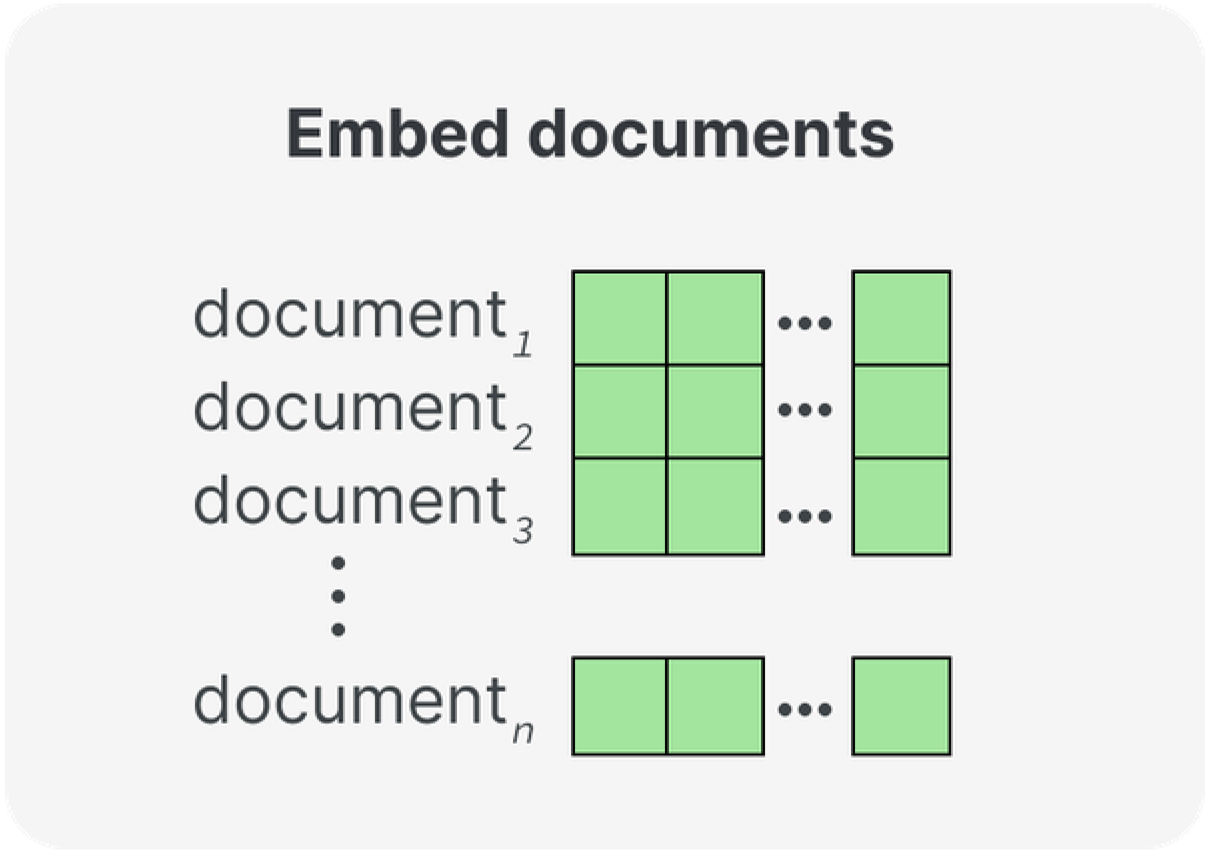


Figure 3-3. Step 1: We convert documents to numerical representations, namely embeddings.

Sentence-transformers has a clear API and can be used as follows to generate embeddings from pieces of text:

```
from sentence_transformers import SentenceTransf

# We load our model
embedding_model = SentenceTransformer('all-MiniL

# The abstracts are converted to vector represen
embeddings = model.encode(abstracts)
```

The sizes of these embeddings differ depending on the model but typically contain at least 384 values for each sentence or paragraph. The number of values an embedding contains is referred to as the dimensionality of the embedding.

problem with dimensionality ?

2. Reduce dimensionality

Before we cluster the embeddings we generated from the ArXiv abstracts, we need to take care of the curse of dimensionality first. This curse is a phenomenon that occurs when dealing with high-dimensional data. As the number of dimensions increases, there is an exponential growth of the number of possible values within each dimension. Finding all subspaces within each dimension becomes increasingly complex. Moreover, as the number of dimensions grows, the concept of distance between points becomes increasingly less precise.

As a result, high-dimensional data can be troublesome for many clustering techniques as it gets more difficult to identify meaningful clusters. Clusters are more diffuse and less distinguishable, making it difficult to accurately identify and separate them.

The previously generated embeddings are high in their dimensionality and often trigger the curse of dimensionality. To prevent their dimensionality from becoming an issue, the second step in our clustering pipeline is dimensionality reduction, as shown in Figure 3-4.

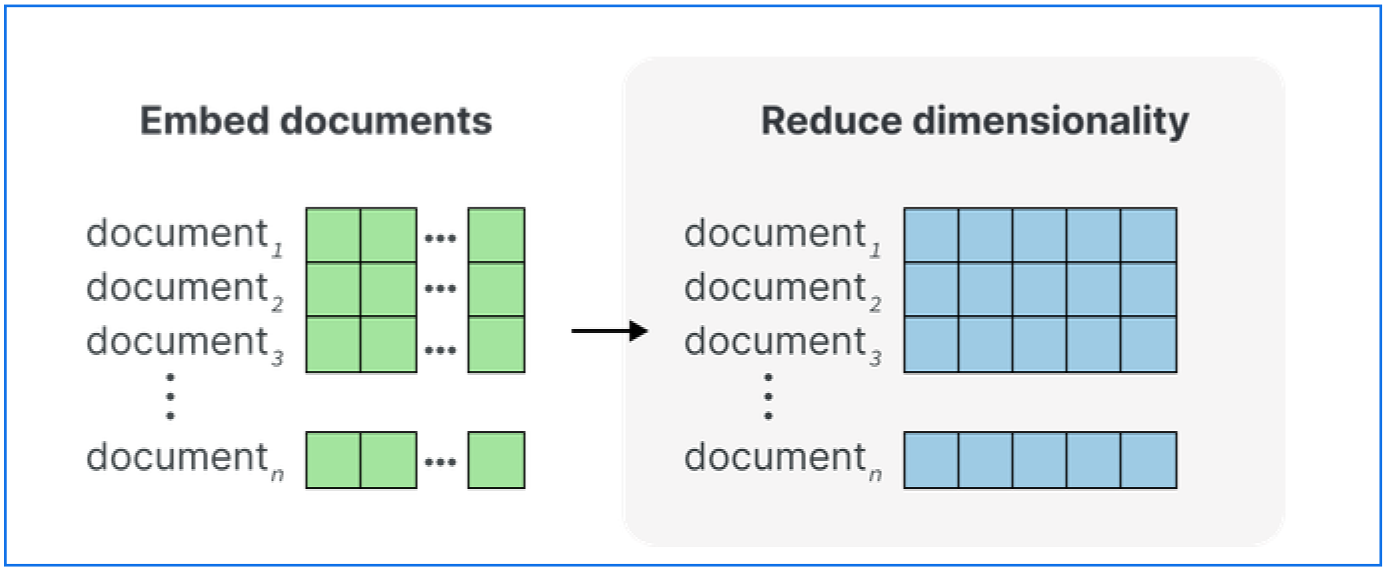


Figure 3-4. Step 2: The embeddings are reduced to a lower dimensional space using dimensionality reduction.

Dimensionality reduction techniques aim to preserve the global structure of high-dimensional data by finding low-dimensional representations. Well-known methods are Principal Component Analysis (PCA) and Uniform Manifold Approximation and Projection (UMAP; mcinnes2018umap). For this pipeline, we are going with UMAP as it tends to handle non-linear relationships and structures a bit better than PCA.

NOTE

Dimensionality reduction techniques, however, are not flawless. They cannot perfectly capture high-dimensional data in a lower-dimensional representation. Information will always be lost with this procedure. There is a balance between reducing dimensionality and keeping as much information as possible.

To perform dimensionality reduction, we need to instantiate our UMAP class and pass the generated embeddings to it:

```
from umap import UMAP

# We instantiate our UMAP model
umap_model = UMAP(n_neighbors=15, n_components=5

# We fit and transform our embeddings to reduce 
reduced_embeddings = umap_model.fit_transform(em
```

We can use the `` `n_components` `` parameter to decide the shape of the lower-dimensional space. Here, we used `` `n_components=5` `` as we want to retain as much information as possible without running into the curse of dimensionality. No one value does this better than another, so feel free to experiment!

3. Cluster embeddings

As shown in Figure 3-5, the final step in our pipeline is to cluster the previously reduced embeddings. Many algorithms out there handle clustering tasks quite well, from centroid-based methods like k-Means to hierarchical methods like Agglomerative Clustering. The choice is up to the user and is highly influenced by the respective use case. Our data might contain some noise, so a clustering algorithm that detects outliers would be preferred. If our data comes in daily, we might want to look for an online or incremental approach instead to model if new clusters were created.

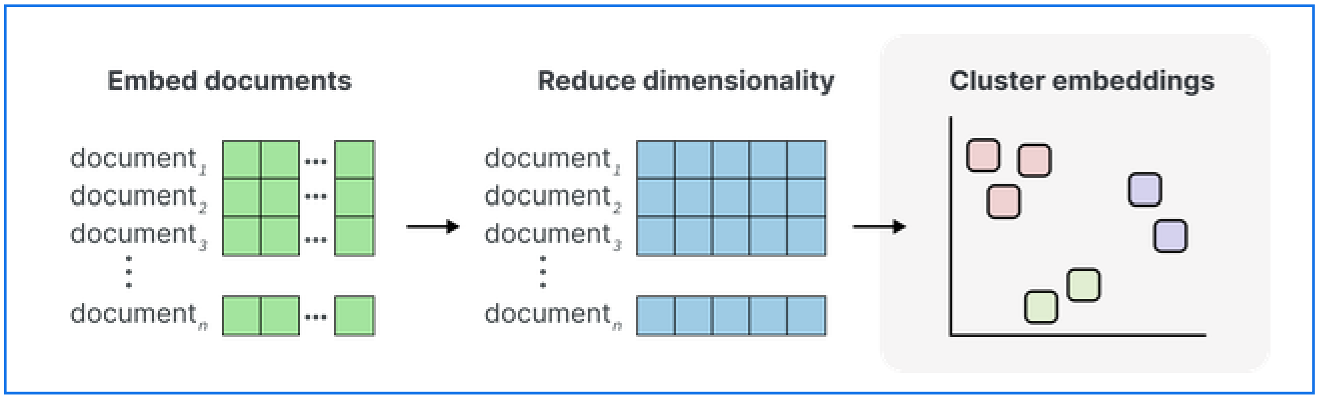


Figure 3-5. Step 3: We cluster the documents using the embeddings that were reduced in their dimensionality.

what is our HDBSCAN

A good default model is Hierarchical Density-Based Spatial Clustering of Applications with Noise (HDBSCAN; mcinnes2017hdbscan). HDBSCAN is a hierarchical variation of a clustering algorithm called DBSCAN which allows for dense (micro)-clusters to be found without us having to explicitly specify the number of clusters. As a density-based method, it can also detect outliers in the data. Data points that do not belong to any cluster. This is important as forcing data into clusters might create noisy aggregations.

As with the previous packages, using HDBSCAN is straightforward. We only need to instantiate the model and pass our reduced embeddings to it:

```
from hdbscan import HDBSCAN

# We instantiate our HDBSCAN model
hdbscan_model = HDBSCAN(min_cluster_size=15, met

# We fit our model and extract the cluster label
hdbscan_model.fit(reduced_embeddings)
labels = hdbscan_model.labels_
```

Then, using our previously generated 2D-embeddings, we can visualize how HDBSCAN has clustered our data:

```
import seaborn as sns

# Reduce 384-dimensional embeddings to 2 dimensi
reduced_embeddings = UMAP(n_neighbors=15, n_comp
min_dist=0.0, metric='cosine').fit_transform(emb
df = pd.DataFrame(np.hstack([reduced_embeddings,
        columns=["x", "y", "cluster"]).sort_values(

# Visualize clusters
df.cluster = df.cluster.astype(int).astype(str)


sns.scatterplot(data=df, x='x', y='y', hue='clus
    linewidth=0, legend=False, s=3, alpha=0.3)
```

As we can see in Figure 3-6, it tends to capture major clusters quite well. Note how clusters of points are colored in the same color, indicating that HDBSCAN put them in a group together. Since we have a large number of clusters, the plotting library cycles the colors between clusters, so don’t think that all blue points are one cluster, for example.

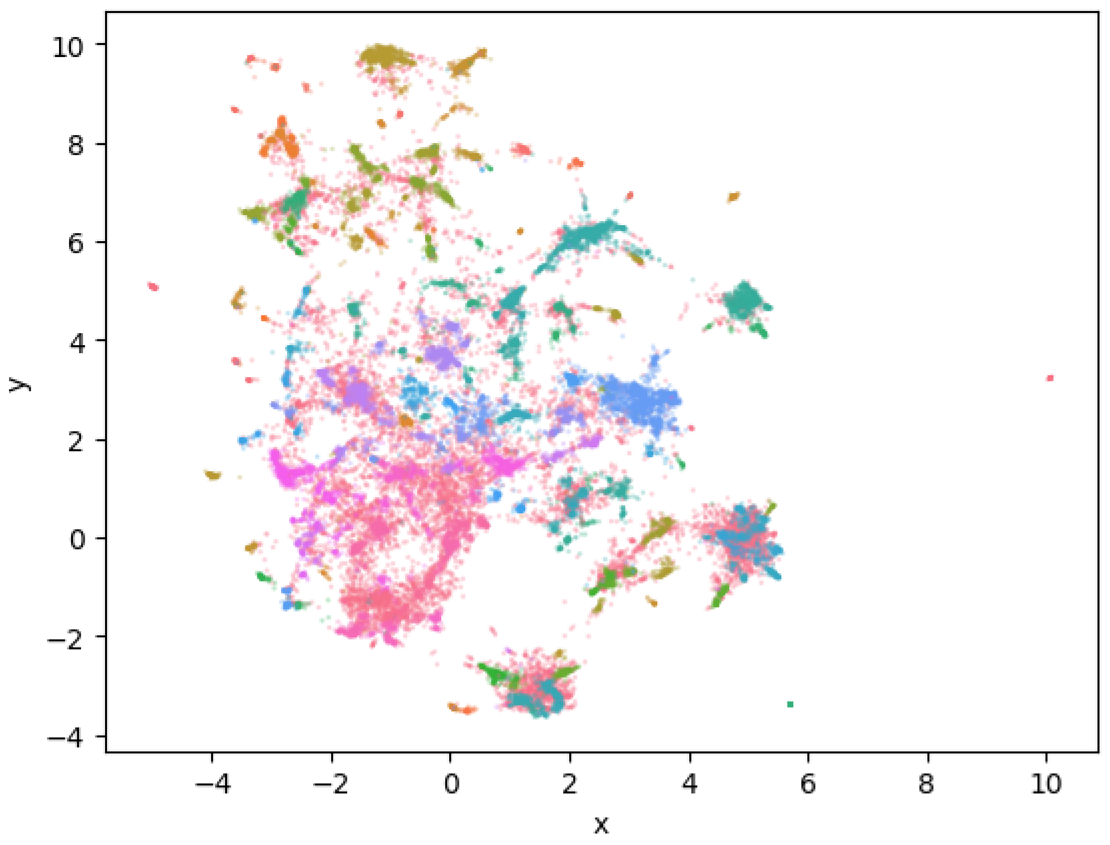


Figure 3-6. The generated clusters (colored) and outliers (grey) are represented as a 2D visualization.

NOTE

Using any dimensionality reduction technique for visualization purposes creates information loss. It is merely an approximation of what our original embeddings look like. Although it is informative, it might push clusters together and drive them further apart than they actually are. Human evaluation, inspecting the clusters ourselves, is, therefore, a key component of cluster analysis!

We can inspect each cluster manually to see which documents are semantically similar enough to be clustered together. For example, let us take a few random documents from cluster **XXX**:

```
>>> for index in np.where(labels==1)[0][:3]:
>>>     print(abstracts[index])
 Sarcasm is considered one of the most difficult
analysis. In our ob-servation on Indonesian soci
people tend to criticize something using sarcasm
additional features to detect sarcasm after a co
con...

  Automatic sarcasm detection is the task of pre
is a crucial step to sentiment analysis, conside
of sarcasm in sentiment-bearing text. Beginning 
speech-based features, sarcasm detection has wit

  We introduce a deep neural network for automat
work has emphasized the need for models to capit
beyond lexical and syntactic cues present in utt
speakers will tend to employ sarcasm regarding d
```

These printed documents tell us that the cluster likely contains documents that talk about **XXX**. We can do this for every created cluster out there but that can be quite a lot of work, especially if we want to experiment with our hyperparameters. Instead, we would like to create a method for automatically extracting representations from these clusters without us having to go through all documents.

This is where topic modeling comes in. It allows us to model these clusters and give singular meaning to them. Although there are many techniques out there, we choose a method that builds upon this clustering philosophy as it allows for significant flexibility.

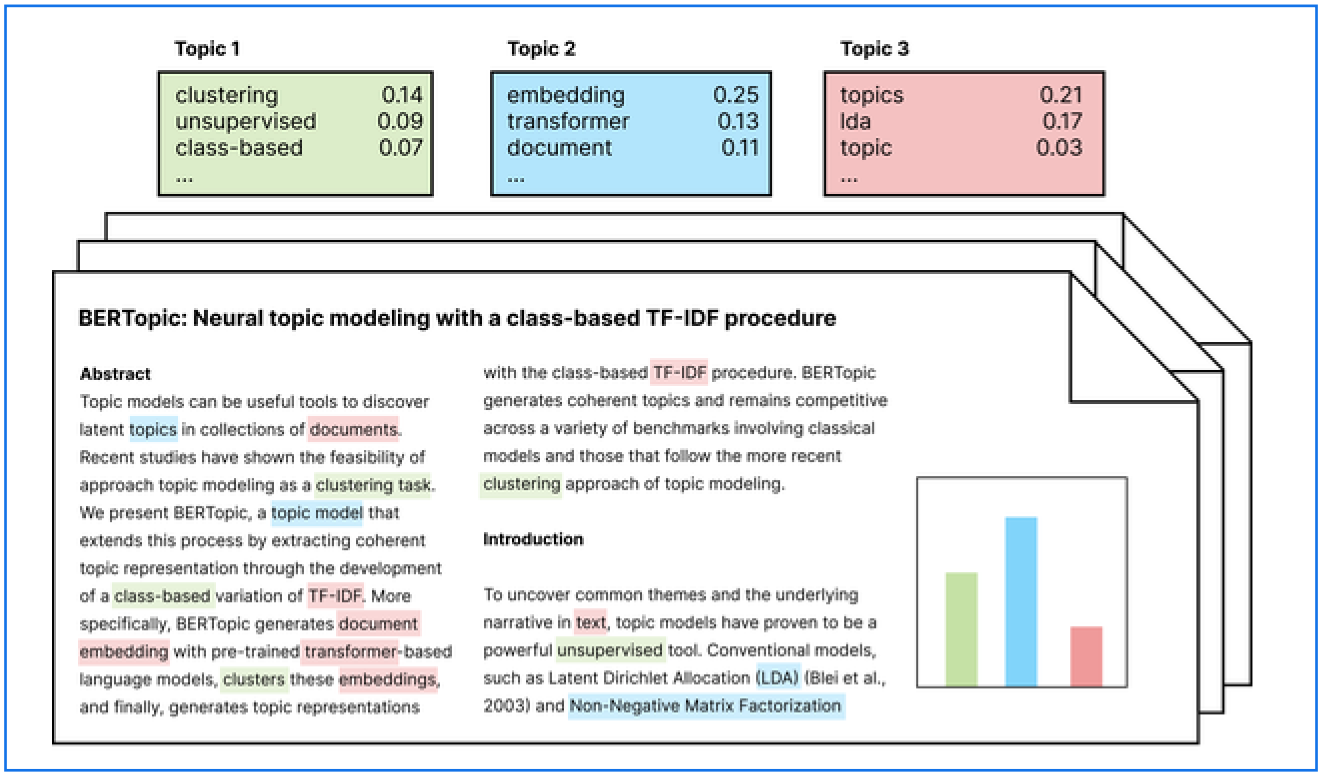
Topic Modeling

what is topic modeling?

Traditionally, topic modeling is a technique that aims to find latent topics or themes in a collection of textual data. For each topic, a set of keywords or phrases are identified that best represent and capture the meaning of the topic. This technique is ideal for finding common themes in large corpora as it gives meaning to sets of similar content. An illustrated overview of topic modeling in practice can be found in Figure 3-7.

Latent Dirichlet Allocation (LDA; blei2003latent) is a classical and popular approach to topic modeling that assumes that each topic is characterized by a probability distribution over words in a corpus vocabulary. Each document is to be considered a mixture of topics. For example, a document about Large

Language Models might have a high probability of containing words like “BERT”, “self-attention”, and “transformers”, while a document about reinforcement learning might have a high probability of containing words like “PPO”, “reward”, “rlhf”.



Topic models can be useful tools to discover latent topics in collections of documents. Recent studies have shown the feasibility of approach topic modeling as a clustering task. We present BERTopic, a topic model that extends this process by extracting coherent topic representation through the development of a class-based variation of TF-IDF. More specifically, BERTopic generates document embedding with pre-trained transformer-based language models, clusters these embeddings, and finally, generates topic representations with the class-based TF-IDF procedure. BERTopic generates coherent topics and remains competitive across a variety of benchmarks involving classical models and those that follow the more recent clustering approach of topic modeling.

To uncover common themes and the underlying narrative in text, topic models have proven to be a powerful unsupervised tool. Conventional models, such as Latent Dirichlet Allocation (LDA) (Blei et al., 2003) and Non-Negative Matrix Factorization

Figure 3-7. An overview of traditional topic modeling.

To this day, the technique is still a staple in many topic modeling use cases, and with its strong theoretical background and practical applications, it is unlikely to go away soon. However, with the seemingly exponential growth of Large Language Models, we start to wonder if we can leverage these Large Language Models in the domain of topic modeling.

There have been several models adopting Large Language Models for topic modeling, like the embedded topic model and the contextualized topic model. However, with the rapid developments in natural language processing, these models have a hard time keeping up.

A solution to this problem is BERTopic, a topic modeling technique that leverages a highly-flexible and modular architecture. Through this modularity, many newly released models can be integrated within its architecture. As the field of Large Language Modeling grows, so does BERTopic. This allows for some interesting and unexpected ways in which these models can be applied in topic modeling.

BERTopic

what is BertTopic?

BERTopic is a topic modeling technique that assumes that clusters of semantically similar documents are a powerful way of generating and describing clusters. The documents in each cluster are expected to describe a major theme and combined they might represent a topic.

As we have seen with text clustering, a collection of documents in a cluster might represent a common theme but the theme itself is not yet described. With text clustering, we would have to go through every single document in a cluster to understand what the cluster is about. To get to the point where we can call a cluster a topic, we need a method for describing that cluster in a condensed and human-readable way.

Although there are quite a few methods for doing so, there is a trick in BERTopic that allows it to quickly describe a cluster, and therefore make it a topic, whilst generating a highly modular pipeline. The underlying algorithm of BERTopic contains, roughly, two major steps.

First, as we did in our text clustering example, we embed our documents to create numerical representations, then reduce their dimensionality and finally cluster the reduced embeddings. The result is clusters of semantically similar documents.

Figure 3-8 describes the same steps as before, namely using sentence-transformers for embedding the documents, UMAP for dimensionality reduction, and HDBSCAN for clustering.

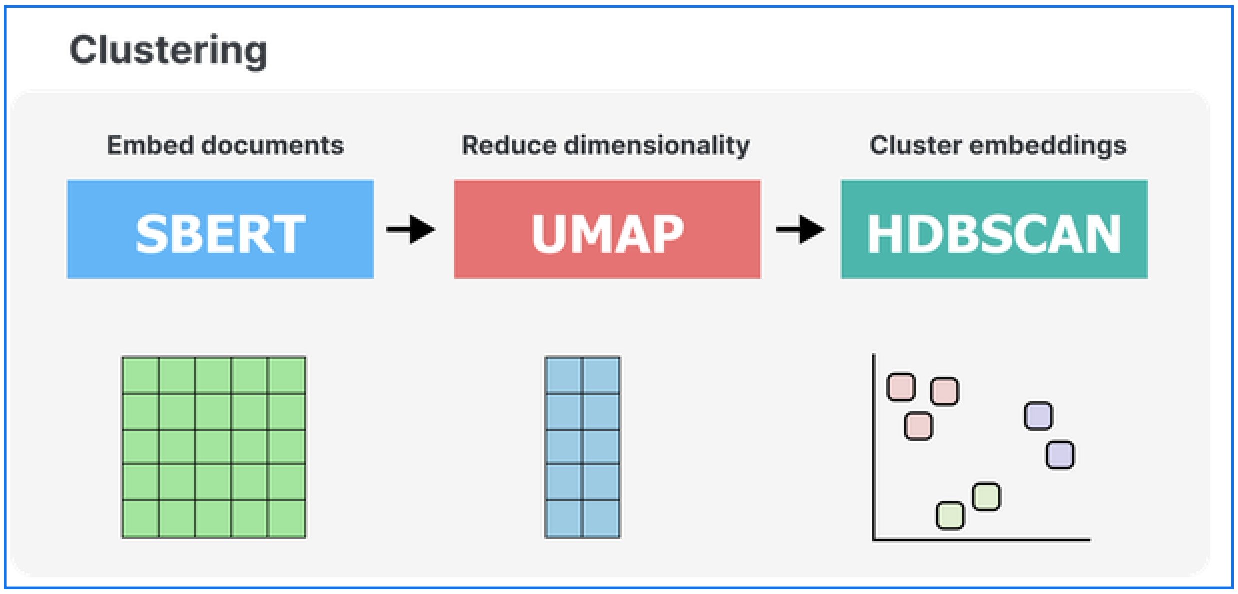
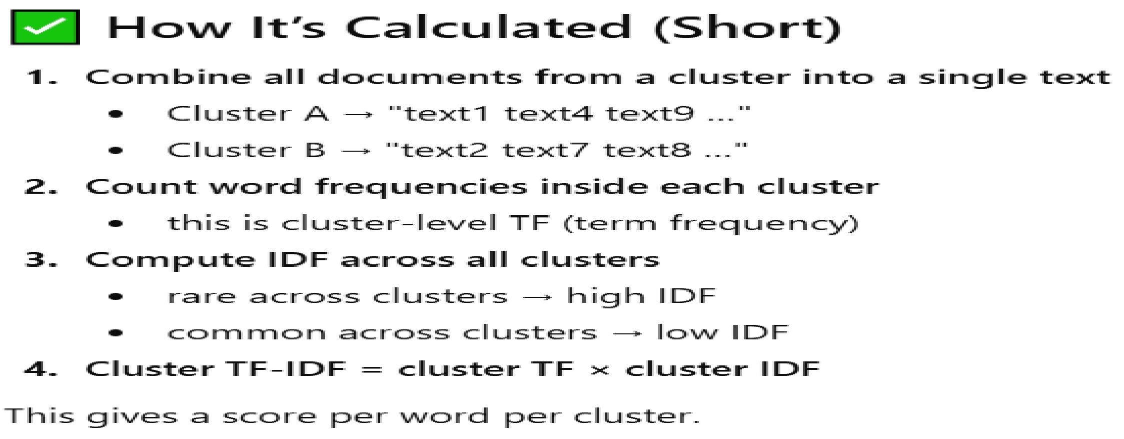


Figure 3-8. The first part of BERTopic's pipeline is clustering textual data.

Second, we find the best-matching keywords or phrases for each cluster. Most often, we would take the centroid of a cluster and find words, phrases, or even sentences that might represent it best. There is a disadvantage to this however: we would have to continuously keep track of our embeddings, and if we were to have millions of documents storing and keeping track becomes computationally difficult. Instead, BERTopic uses the classic bag-of-words method to represent the clusters. A bag of words is exactly what the name implies, for each document we simply count how often a certain word appears and use that as our textual representation.

However, words like "the", "and", and "I" appear quite frequently in most English texts and are likely to be overrepresented. To give proper weight to these words, BERTopic uses a technique called c-TF-IDF, which stands for class-based term-frequency inverse-document frequency. c-TF-IDF is a class-based adaptation of the classic TF-IDF procedure. Instead of considering the importance of words within documents, c-TF-IDF considers the importance of words between clusters of documents.

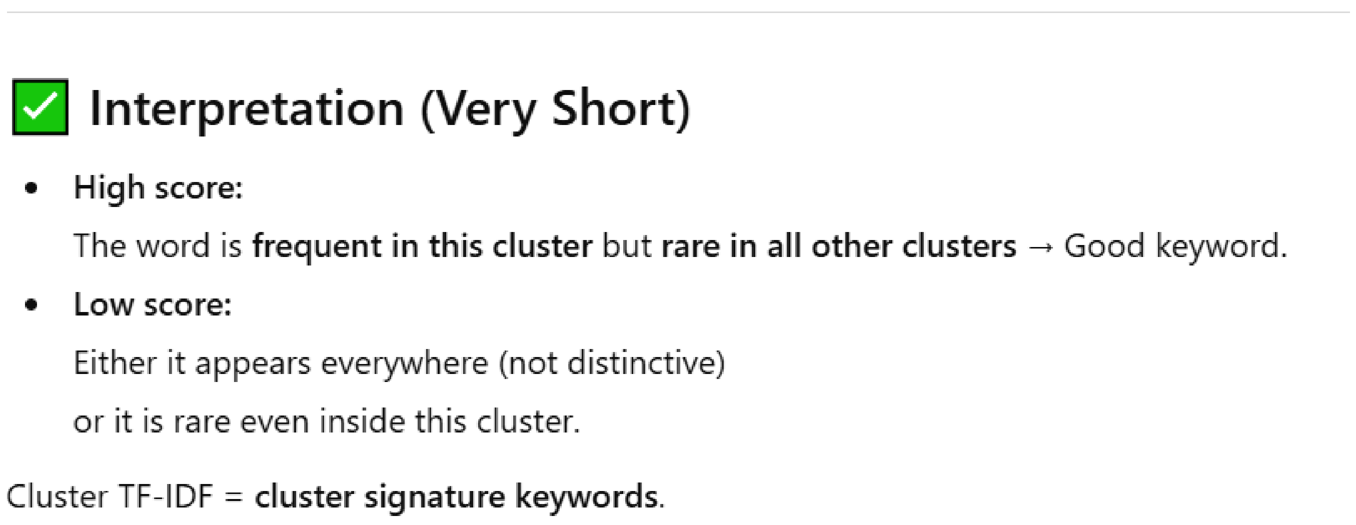
To use c-TF-IDF, we first concatenate each document in a cluster to generate one long document. Then, we extract the frequency of the term *f_x* in class *c*, where *c* refers to one of the clusters we created before. Now we have, per cluster, how many and which words they contain, a mere count.



✅ How It's Calculated (Short)

1. **Combine all documents from a cluster into a single text**
   - Cluster A → "text1 text4 text9 ..."
   - Cluster B → "text2 text7 text8 ..."
2. **Count word frequencies inside each cluster**
   - this is cluster-level TF (term frequency)
3. **Compute IDF across all clusters**
   - rare across clusters → high IDF
   - common across clusters → low IDF
4. **Cluster TF-IDF = cluster TF × cluster IDF**

This gives a score per word per cluster.

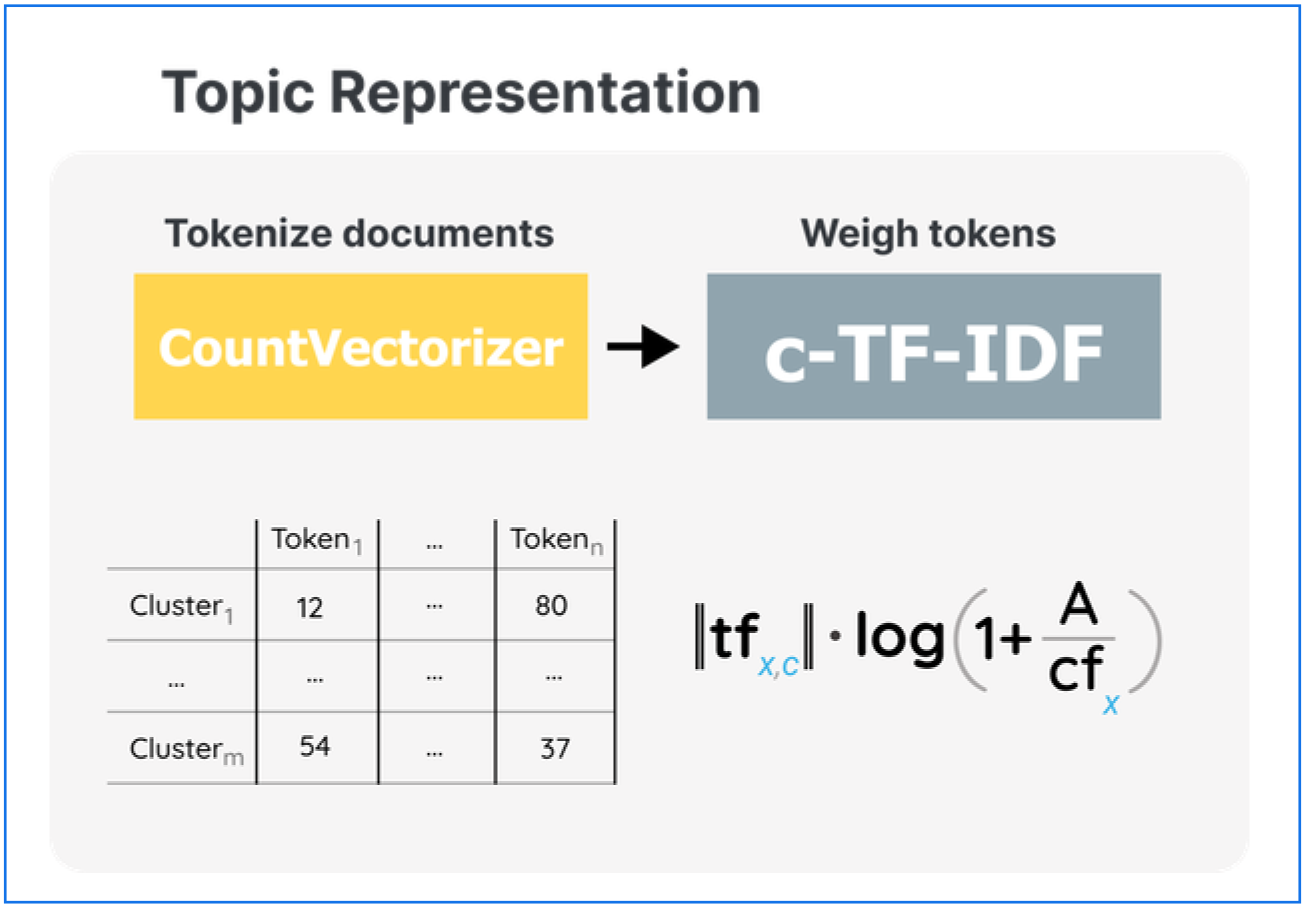


✅ Interpretation (Very Short)

- **High score:**
  The word is **frequent in this cluster** but **rare in all other clusters** → Good keyword.
- **Low score:**
  Either it appears everywhere (not distinctive)
  or it is rare even inside this cluster.

Cluster TF-IDF = **cluster signature keywords**.

contain. It is essentially a ranking of a corpus' vocabulary in each topic.



	Token_1	...	Token_n
Cluster_1	12	...	80
...	...	...	...
Cluster_m	54	...	37

$$\|tf_{x,c}\| \cdot \log\left(1 + \frac{A}{cf_x}\right)$$

*Figure 3-9. The second part of BERTopic's pipeline is representing the topics. The calculation of the weight of term *x* in a class *c*.*

Putting the two steps together, clustering and representing topics, results in the full pipeline of BERTopic, as illustrated in Figure 3-10. With this pipeline, we can cluster semantically similar documents and from the clusters generate topics represented by several keywords. The higher the weight of a keyword for a topic, the more representative it is of that topic.

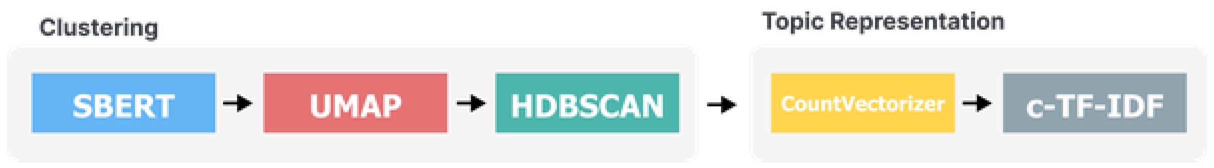


Figure 3-10. The full pipeline of BERTopic, roughly, consists of two steps, clustering and topic representation.

NOTE

Interestingly, the c-TF-IDF trick does not use a Large Language Model and therefore does not take the context and semantic nature of words into account. However, like with neural search, it allows for an efficient starting point after which we can use the more compute-heavy techniques, such as GPT-like models.

One major advantage of this pipeline is that the two steps, clustering and topic representation, are relatively independent of one another. When we generate our topics using c-TF-IDF, we do not use the models from the clustering step, and, for example, do not need to track the embeddings of every single document. As a result, this allows for significant modularity not only with respect to the topic generation process but the entire pipeline.

NOTE

With clustering, each document is assigned to only a single cluster or topic. In practice, documents might contain multiple topics, and assigning a multi-topic document to a single topic would not always be the most accurate method. We will go into this later, as BERTopic has a few ways of handling this, but it is important to understand that at its core, topic modeling with BERTopic is a clustering task.

The modular nature of BERTopic's pipeline is extensible to every component. Although sentence-transformers are used as a default embedding model for transforming documents to numerical representations, nothing is stopping us from using any other embedding technique. The same applies to the dimensionality reduction, clustering, and topic generation process. Whether a use case calls for k-Means instead of HDBSCAN, and PCA instead of UMAP, anything is possible.

You can think of this modularity as building with lego blocks, each part of the pipeline is completely replaceable with another, similar algorithm. This "lego block" way of thinking is illustrated in Figure 3-11. The figure also shows an additional algorithmic lego block that we can use. Although we use c-TF-IDF to create our initial topic representations, there are a number of interesting ways we can use LLMs to fine-tune these representations. In the "**Representation Models**" section below, we will go into extensive detail on how this algorithmic lego block works.

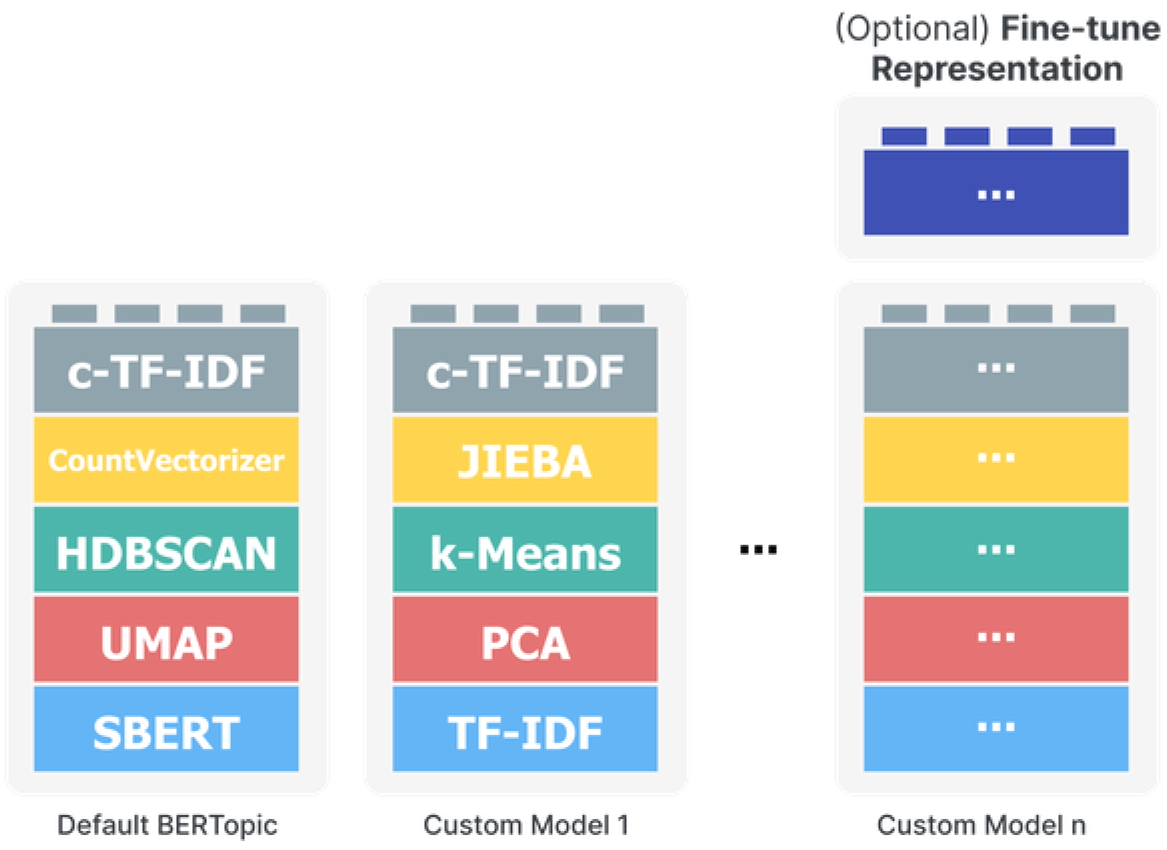


Figure 3-11. The modularity of BERTopic is a key component and allows you to build your own topic model whoever you want.

Code Overview

Enough talk! This is a hands-on book, so it is finally time for some hands-on coding. The default pipeline, as illustrated previously in Figure 3-10, only requires a few lines of code:

```
from bertopic import BERTopic

# Instantiate our topic model
topic_model = BERTopic()

# Fit our topic model on a list of documents
topic_model.fit(documents)
```

However, the modularity that BERTopic is known for and that we have visualized thus far can also be visualized through a coding example. First, let us import some relevant packages:

```
from umap import UMAP
from hdbscan import HDBSCAN
from sentence_transformers import SentenceTransfo
from sklearn.feature_extraction.text import Coun

from bertopic import BERTopic
from bertopic.representation import KeyBERTInspi
from bertopic.vectorizers import ClassTfidfTrans
```

As you might have noticed, most of the imports, like UMAP and HDBSCAN, are part of the default BERTopic pipeline. Next, let us build the default pipeline of BERTopic a bit more explicitly and go each individual step:

```
# Step 1 - Extract embeddings (blue block)
embedding_model = SentenceTransformer("all-MiniLM

# Step 2 - Reduce dimensionality (red block)
umap_model = UMAP(n_neighbors=15, n_components=5

# Step 3 - Cluster reduced embeddings (green bloc
hdbscan_model = HDBSCAN(min_cluster_size=15, met

# Step 4 - Tokenize topics (yellow block)
vectorizer_model = CountVectorizer(stop_words="en

# Step 5 - Create topic representation (grey bloc
ctfidf_model = ClassTfidfTransformer()

# Step 6 - (Optional) Fine-tune topic representat
# a `bertopic.representation` model (purple block
representation_model = KeyBERTInspired()
# Combine the steps and build our own topic model

topic_model = BERTopic(
  embedding_model=embedding_model,          # Ste
  umap_model=umap_model,                    # Ste
  hdbscan_model=hdbscan_model,              # Ste
  vectorizer_model=vectorizer_model,        # Ste
  ctfidf_model=ctfidf_model,                # Ste
  representation_model=representation_model # Ste
)
```

This code allows us to go through all steps of the algorithm explicitly and essentially let us build the topic model however we want. The resulting topic model, as defined in the variable `topic_model`, now represents the base pipeline of BERTopic as illustrated back in Figure 3-10.

Example

We are going to keep using the abstracts of ArXiv articles throughout this use case. To recap what we did with text clustering, we start by importing our dataset using HuggingFace's dataset package and extracting metadata that we are going to use later on, like the abstracts, years, and categories of the articles.

```
# Load data from huggingface
from datasets import load_dataset
dataset = load_dataset("maartengr/arxiv_nlp")

# Extract specific metadata
abstracts = dataset["Abstracts"]
years = dataset["Years"]
categories = dataset["Categories"]
titles = dataset["Titles"]
```

Using BERTopic is quite straightforward, and it can be used in just three lines:

```
# Train our topic model in only three lines of c
from bertopic import BERTopic

topic_model = BERTopic()
topics, probs = topic_model.fit_transform(abstra
```

With this pipeline, you will have 3 variables returned, namely `topic_model`, `topics`, and `probs`:

- `topic_model` is the model that we have just trained before and contains information about the model and the topics that we created.
- `topics` are the topics for each abstract.
- `probs` are the probabilities that a topic belongs to a certain abstract.

Before we start to explore our topic model, there is one change that we will need to make the results reproducible. As mentioned before, one of the underlying models of BERTopic is UMAP. This model is stochastic in nature which means that every time we run BERTopic, we will get different results. We can prevent this by passing a `random_state` to the UMAP model.

```
from umap import UMAP
from bertopic import BERTopic

# Using a custom UMAP model
umap_model = UMAP(n_neighbors=15, n_components=5

# Train our model
topic_model = BERTopic(umap_model=umap_model)
topics, probs = topic_model.fit_transform(abstra
```

Now, let's start by exploring the topics that were created. The `get_topic_info()` method is useful to get a quick description of the topics that we found:

```
>>> topic_model.get_topic_info()
Topic    Count    Name
0    -1    11648    -1_of_the_and_to
1    0    1554    0_question_answer_questions_qa
2    1    620    1_hate_offensive_toxic_detectio
3    2    578    2_summarization_summaries_summa
4    3    568    3_parsing_parser_dependency_amr
...    ...    ...    ...
217    216    10    216_prf_search_conversational
```

```
317     316     10     316_prf_search_conversationa
318     317     10     317_crowdsourcing_workers_an
319     318     10     318_curriculum_nmt_translati
320     319     10     319_botsim_menu_user_dialogu
321     320     10     320_color_colors_ib_naming
```

There are many topics generated from our model, **XXX**! Each of these topics is represented by several keywords, which are concatenated with a “_” in the Name column. This Name column allows us to quickly get a feeling of what the topic is about as it shows the four keywords that best represent it.

NOTE

You might also have noticed that the very first topic is labeled -1. That topic contains all documents that could not be fitted within a topic and are considered to be outliers. This is a result of the clustering algorithm, HDBSCAN, that does not force all points to be clustered. To remove outliers, we could either use a non-outlier algorithm like k-Means or use BERTopic’s `reduce_outliers()` function to remove some of the outliers and assign them to topics.

For example, topic 2 contains the keywords “summarization”, “summaries”, “summary”, and “abstractive”. Based on these keywords, it seems that the topic is summarization tasks. To get the top 10 keywords per topic as well as their c-TF-IDF weights, we can use the get_topic() function:

```
>>> topic_model.get_topic(2)
[('summarization', 0.029974019692323675),
 ('summaries', 0.018938088406361412),
 ('summary', 0.018019112468622436),
 ('abstractive', 0.015758156442697138),
 ('document', 0.011038627359130419),
 ('extractive', 0.010607624721836042),
 ('rouge', 0.009363770589253410),
 ('factual', 0.005651676100789188),
 ('sentences', 0.005262910357048789),
 ('mds', 0.005050565343932314)]
```

This gives us a bit more context about the topic and helps us understand what the topic is about. For example, it is interesting to see the word “rogue” appear since that is a common metric for evaluating summarization models.

We can use the `find_topics()` function to search for specific topics based on a search term. Let’s search for a topic about topic modeling:

```
>>> topic_model.find_topics("topic modeling")
([17, 128, 116, 6, 235],
 [0.6753638370140129,
  0.40951682679389345,
  0.3985390076544335,
    0.37922002441932795,
    0.3769700288091359])
```

It returns that topic 17 has a relatively high similarity (0.675) with our search term. If we then inspect the topic, we can see that it is indeed a topic about topic modeling:

```
>>> topic_model.get_topic(17)
[('topic', 0.0503756681079549),
 ('topics', 0.02834246786579726),
 ('lda', 0.015441277604137684),
 ('latent', 0.011458141214781893),
 ('documents', 0.01013764950401255),
 ('document', 0.009854201885298964),
 ('dirichlet', 0.009521114618288628),
 ('modeling', 0.008775384549157435),
 ('allocation', 0.0077508974418589605),
 ('clustering', 0.005909325849593925)]
```

Although we know that his topic is about topic modeling, let us see if the BERTopic abstract is also assigned to this topic:

```
>>> topics[titles.index('BERTopic: Neural topic 
17
```

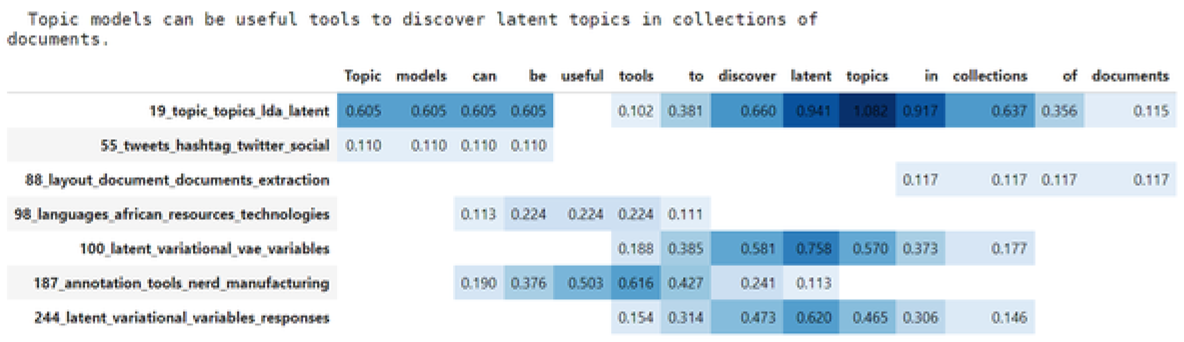
It is! It seems that the topic is not just about LDA-based methods but also cluster-based techniques, like BERTopic.

Lastly, we mentioned before that many topic modeling techniques assume that there can be multiple topics within a single document or even a sentence. Although BERTopic leverages clustering, which assumes a single assignment to each data point, it can approximate the topic distribution.

We can use this technique to see what the topic distribution is of the first sentence in the BERTopic paper:

```
index = titles.index('BERTopic: Neural topic mod

# Calculate the topic distributions on a token-le
topic_distr, topic_token_distr = topic_model.app
df = topic_model.visualize_approximate_distribut
df
```



```
 Topic models can be useful tools to discover latent topics in collections of
documents.
```

	Topic	models	can	be	useful	tools	to	discover	latent	topics	in	collections	of	documents
19_topic_topics_lda_latent	0.605	0.605	0.605	0.605		0.102	0.381	0.660	0.941	1.082	0.917	0.637	0.356	0.115
55_tweets_hashtag_twitter_social	0.110	0.110	0.110	0.110										
88_layout_document_documents_extraction											0.117	0.117	0.117	0.117
98_languages_african_resources_technologies			0.113	0.224	0.224	0.224	0.111							
100_latent_variational_vae_variables						0.188	0.385	0.581	0.758	0.570	0.373	0.177		
187_annotation_tools_nerd_manufacturing			0.190	0.376	0.503	0.616	0.427	0.241	0.113					
244_latent_variational_variables_responses						0.154	0.314	0.473	0.620	0.465	0.306	0.146		

Figure 3-12. A wide range of visualization options are available in BERTopic.

The output, as shown in Figure 3-12, demonstrates that the document, to a certain extent, contains multiple topics. This assignment is even done on a token level!

(Interactive) Visualizations

Going through **XXX** topics manually can be quite a task. Instead, several helpful visualization functions allow us to get a broad overview of the topics that were generated. Many of which are interactive by using the Plotly visualization framework.

Figure 3-13 shows all possible visualization options in BERTopic, from 2D document representations and topic bar charts to topic hierarchy and similarity. Although we are not going through all visualizations, there are some worth looking into.

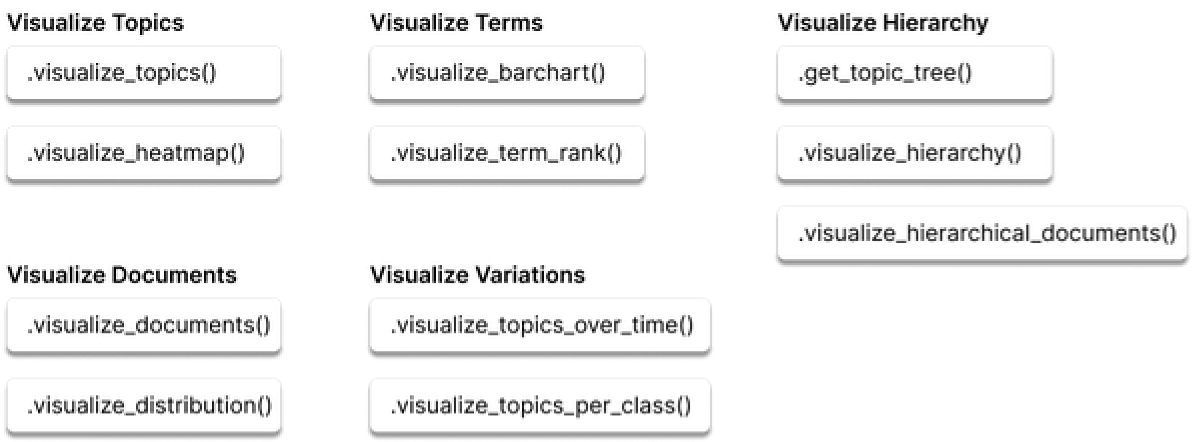


Figure 3-13. A wide range of visualization options are available in BERTopic.

To start, we can create a 2D representation of our topics by using UMAP to reduce the c-TF-IDF representations of each topic.

```
topic_model.visualize_topics()
```

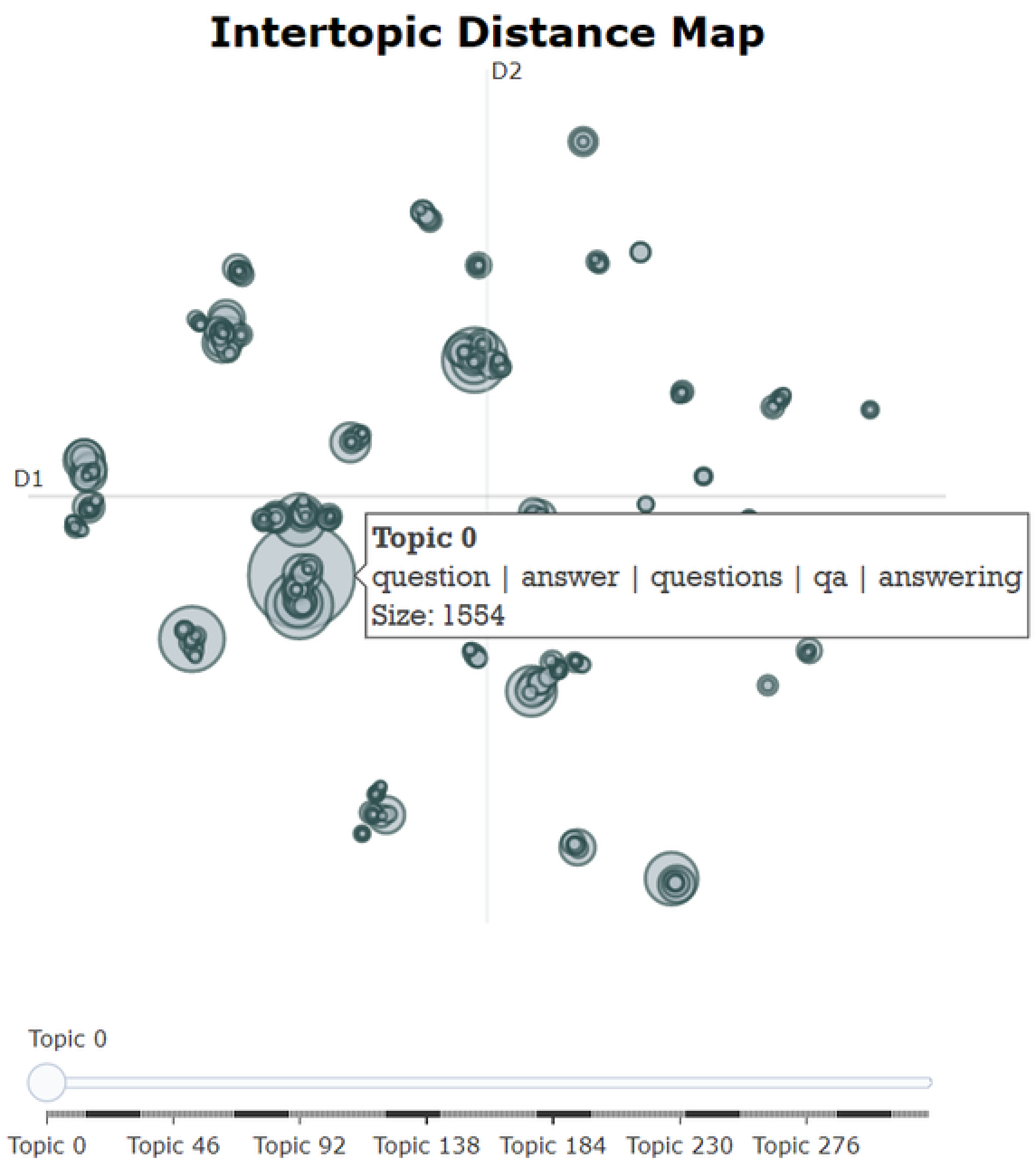


Figure 3-14. The intertopic distance map of topics represented in 2D space.

As shown in Figure 3-14, this generates an interactive visualization that, when hovering over a circle, allows us to see the topic, its keywords, and its size. The larger the circle of a topic is, the more documents it contains. We can quickly see groups of similar topics through interaction with this visualization.

We can use the `visualize_documents()` function to take this analysis to the next level, namely analyzing topics on a document level.

```
# Visualize a selection of topics and documents
topic_model.visualize_documents(titles,
      topics=[0, 1, 2, 3, 4, 6, 7, 10, 12,
 13, 16, 33, 40, 45, 46, 65])
```

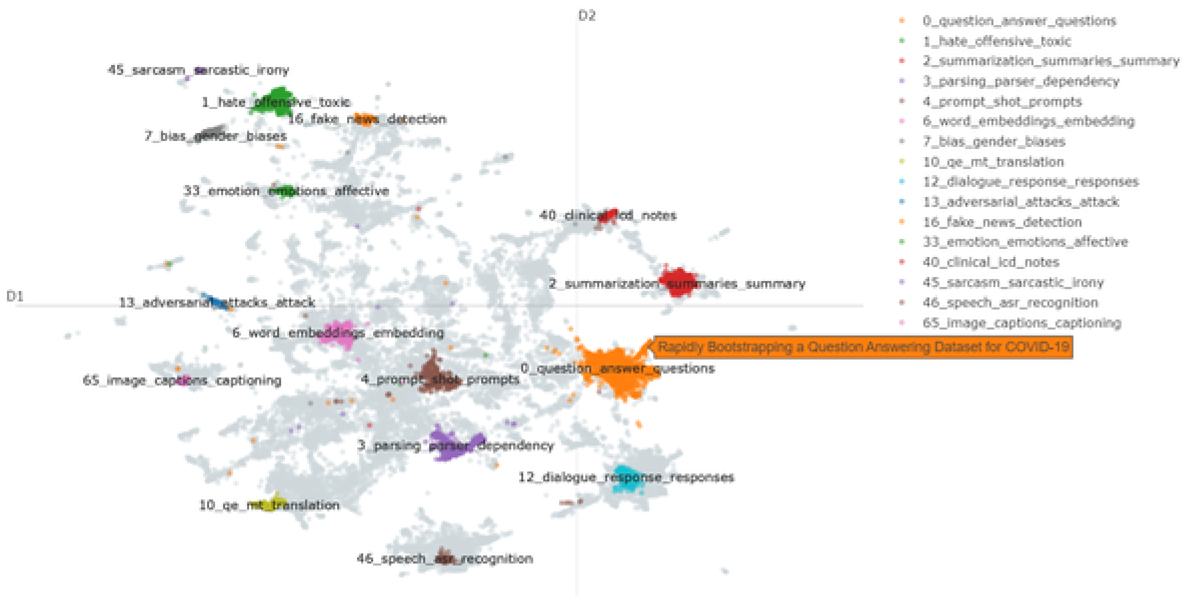


Figure 3-15. Abstracts and their topics are represented in a 2D visualization.

Figure 3-15 demonstrates how BERTopic can visualize documents in a 2D-space.

NOTE

We only visualized a selection of topics since showing all 300 topics would result in quite a messy visualization. Also, instead of passing `abstracts`, we passed `titles` since we only want to view the titles of each paper when we hover over a document and not the entire abstract.

Lastly, we can create a bar chart of the keywords in a selection of topics using visualize_barchart():

```
topic_model.visualize_barchart(topics=list(range
```

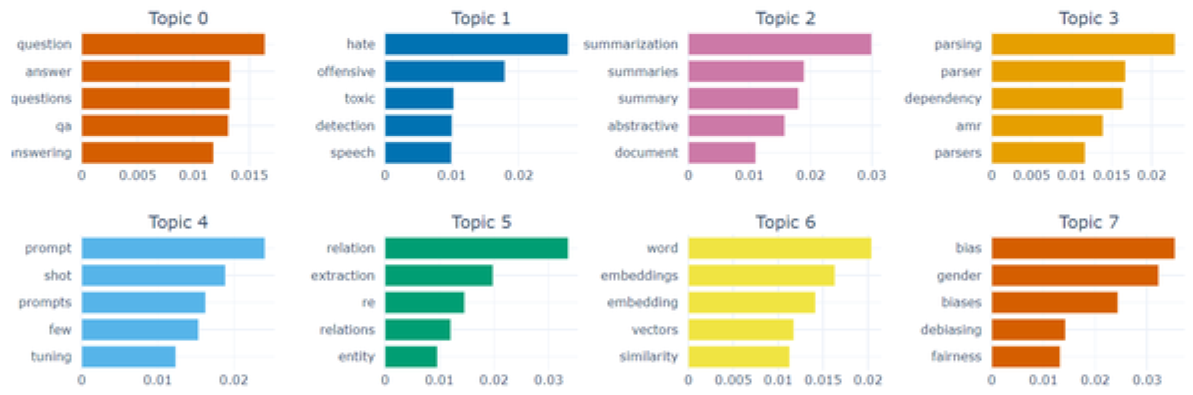


Figure 3-16. The top 5 keywords for the first 8 topics.

The bar chart in Figure 3-16 gives a nice indication of which keywords are most important to a specific topic. Take topic 2 for example–it seems that the word “summarization” is most representative of that topic and that other words are very similar in importance.

Representation Models

With the neural-search style modularity that BERTopic employs, it can leverage many different types of Large Language Models whilst minimizing computing. This allows for a large range of topic fine-tuning methods, from part-of-speech to text-generation methods, like ChatGPT. Figure 3-17 demonstrates the variety of LLMs that we can leverage to fine-tune topic representations.

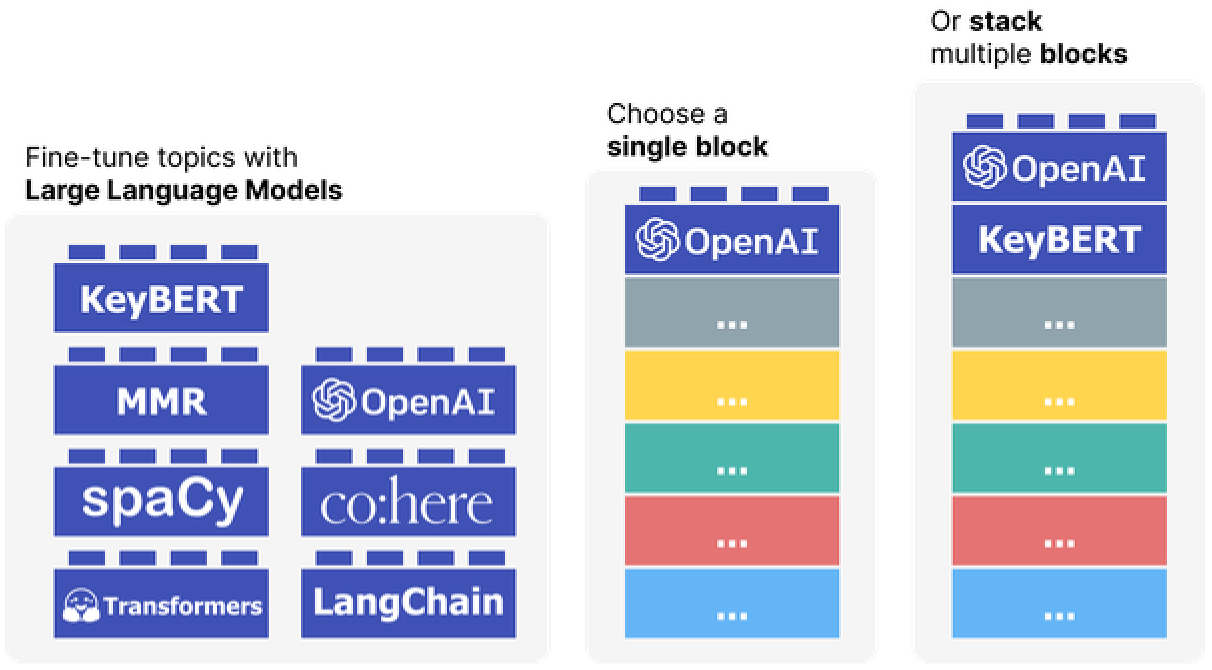

Fine-tune topics with **Large Language Models**
KeyBERT
MMR
spaCy
Transformers
OpenAI
co:here
LangChain
Choose a **single block**
OpenAI
...
Or **stack** multiple **blocks**
OpenAI
KeyBERT
...

Figure 3-17. After applying the c-TF-IDF weighting, topics can be fine-tuned with a wide variety of representation models. Many of which are Large Language Models.

Topics generated with c-TF-IDF serve as a good first ranking of words with respect to their topic. In this section, these initial rankings of words can be considered candidate keywords for a topic as we might change their rankings based on any representation model. We will go through several representation models that can be used within BERTopic and that are also interesting from a Large Language Modeling standpoint.

Before we start, we first need to do two things. First, we are going to save our original topic representations so that it will be much easier to compare with and without representation models:

```
# Save original representations
from copy import deepcopy
original_topics = deepcopy(topic_model.topic_rep
```

Second, let’s create a short wrapper that we can use to quickly visualize the differences in topic words to compare with and without representation models:

```
def topic_differences(model, original_topics, ma
""" For the first 10 topics, show the differen
topic representations between two models """
for topic in range(nr_topics):

  # Extract top 5 words per topic per model

  og_words = " | ".join(list(zip(*original_top
  new_words = " | ".join(list(zip(*model.get_t

  # Print a 'before' and 'after'
  whitespaces = " " * (max_length - len(og_wor
  print(f"Topic: {topic}    {og_words}{whitespa
```

KeyBERTInspired

c-TF-IDF generated topics do not consider the semantic nature of words in a topic which could end up creating topics with stopwords. We can use the module **bertopic.representation_model.KeyBERTInspired()** to fine-tune the topic keywords based on their semantic similarity to the topic.

KeyBERTInspired is, as you might have guessed, a method inspired by the keyword extraction package, KeyBERT. In its most basic form, KeyBERT compares the embeddings of words in a document with the document embedding using cosine similarity to see which words are most related to the document. These most similar words are considered keywords.

In BERTopic, we want to use something similar but on a topic level and not a document level. As shown in Figure 3-18, KeyBERTInspired uses c-TF-IDF to create a set of representative documents per topic by randomly sampling 500 documents per topic, calculating their c-TF-IDF values, and finding the most representative documents. These documents are embedded and averaged to be used as an updated topic embedding. Then, the similarity between our candidate keywords and the updated topic embedding is calculated to re-rank our candidate keywords.

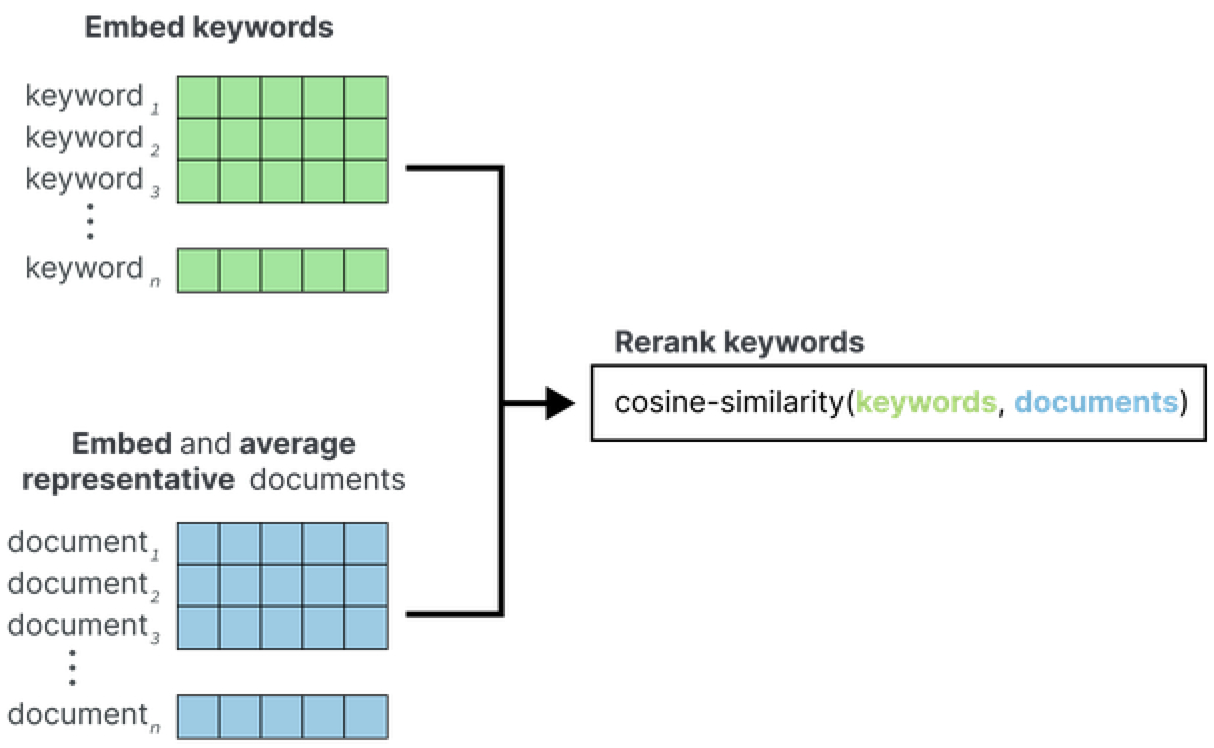


Figure 3-18. The procedure of the KeyBERTInspired representation model

```
# KeyBERTInspired
from bertopic.representation import KeyBERTInspi
representation_model = KeyBERTInspired()

# Update our topic representations
new_topic_model.update_topics(abstracts, represe

# Show topic differences
topic_differences(topic_model, new_topic_model)
```

```
Topic: 0 question | qa | questions | answer |
answering --> questionanswering | answering |
questionanswer | attention | retrieval
```

```
Topic: 1 hate | offensive | speech | detection |
toxic --> hateful | hate | cyberbullying | speech
| twitter
```

```
Topic: 2 summarization | summaries | summary |
abstractive | extractive --> summarizers |
summarizer | summarization | summarisation |
summaries
```

```
Topic: 3 parsing | parser | dependency | amr |
parsers --> parsers | parsing | treebanks | parser
| treebank
```

```
Topic: 4 word | embeddings | embedding |
similarity | vectors --> word2vec | embeddings |
embedding | similarity | semantic
```

```
Topic: 5 gender | bias | biases | debiasing |
fairness --> bias | biases | genders | gender |
gendered
```

```
Topic: 6 relation | extraction | re | relations |
entity --> relations | relation | entities |
entity | relational
```

```
Topic: 7 prompt | fewshot | prompts | incontext |
tuning --> prompttuning | prompts | prompt |
prompting | promptbased
```

```
Topic: 8 aspect | sentiment | absa | aspectbased |
opinion --> sentiment | aspect | aspects |
aspectlevel | sentiments
```

```
Topic: 9 explanations | explanation | rationales |
rationale | interpretability --> explanations |
explainers | explainability | explaining |
attention
```

The updated model shows that the topics are much easier to read compared to the original model. It also shows the downside of using embedding-based techniques. Words in the original model, like "amr" and "qa" are perfectly reasonable words

Part-of-Speech

c-TF-IDF does not make any distinction of the type of words it deems to be important. Whether it is a noun, verb, adjective, or even a preposition, they can all end up as important keywords. When we want to have human-readable labels that are straightforward and intuitive to interpret, we might want topics that are described by, for example, nouns only.

This is where the well-known SpaCy package comes in. An industrial-grade NLP framework that comes with a variety of pipelines, models, and deployment options. More specifically, we can use SpaCy to load in an English model that is capable of detecting part of speech, whether a word is a noun, verb, or something else.

As shown in Figure 3-19, we can use SpaCy to make sure that only nouns end up in our topic representations. As with most representation models, this is highly efficient since the nouns are extracted from only a small but representative subset of the data.

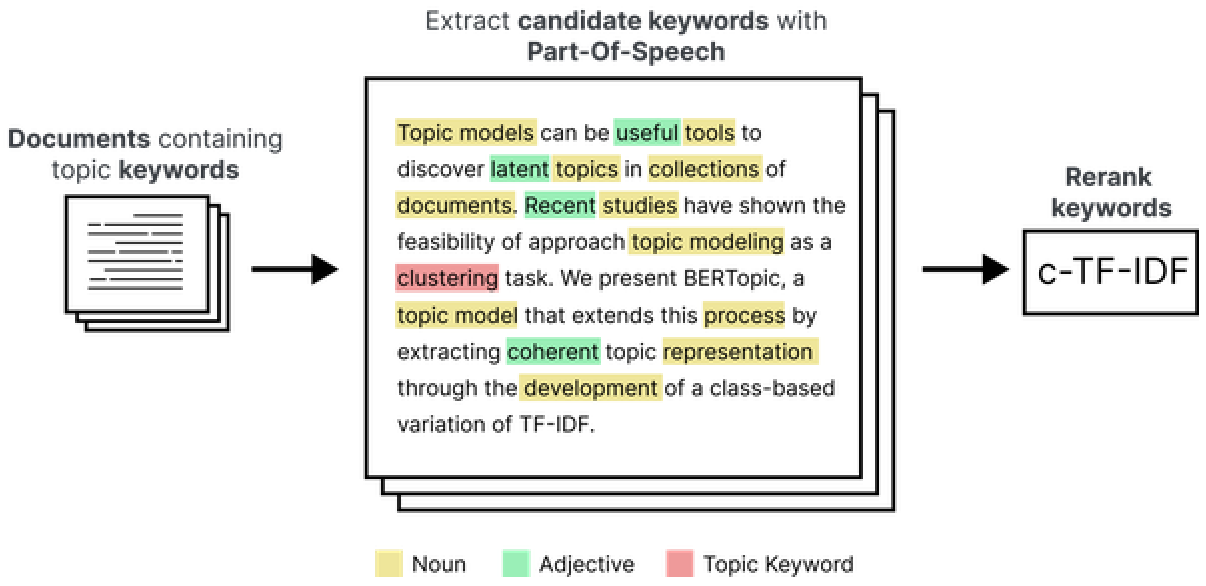

Documents containing topic keywords
Extract candidate keywords with Part-Of-Speech
Topic models can be useful tools to discover latent topics in collections of documents. Recent studies have shown the feasibility of approach topic modeling as a clustering task. We present BERTopic, a topic model that extends this process by extracting coherent topic representation through the development of a class-based variation of TF-IDF.
Rerank keywords
c-TF-IDF
Noun
Adjective
Topic Keyword

Figure 3-19. The procedure of the PartOfSpeech representation model

```
# Part-of-Speech tagging
from bertopic.representation import PartOfSpeech
representation_model = PartOfSpeech("en_core_web

# Use the representation model in BERTopic on to
topic_model.update_topics(abstracts, representat

# Show topic differences
topic_differences(topic_model, original_topics)
```

```
Topic: 0 question | qa | questions | answer | answering --> question | questions | answer | answering | answers

Topic: 1 hate | offensive | speech | detection | toxic --> hate | offensive | speech | detection | toxic

Topic: 2 summarization | summaries | summary | abstractive | extractive --> summarization | summaries | summary | abstractive | extractive

Topic: 3 parsing | parser | dependency | amr | parsers --> parsing | parser | dependency | parsers | treebank
```

```
Topic: 4 word | embeddings | embedding | similarity | vectors --> word | embeddings | similarity | vectors | words

Topic: 5 gender | bias | biases | debiasing | fairness --> gender | bias | biases | debiasing | fairness

Topic: 6 relation | extraction | re | relations | entity --> relation | extraction | relations | entity | distant

Topic: 7 prompt | fewshot | prompts | incontext | tuning --> prompt | prompts | tuning | prompting | tasks

Topic: 8 aspect | sentiment | absa | aspectbased | opinion --> aspect | sentiment | opinion | aspects | polarity

Topic: 9 explanations | explanation | rationales | rationale | interpretability --> explanations | explanation | rationales | rationale | interpretability
```

Maximal Marginal Relevance

With c-TF-IDF, there can be a lot of redundancy in the resulting keywords as it does not consider words like “car” and “cars” to be essentially the same thing. In other words, we want sufficient diversity in the resulting topics with as little repetition as possible. (Figure 3-20)

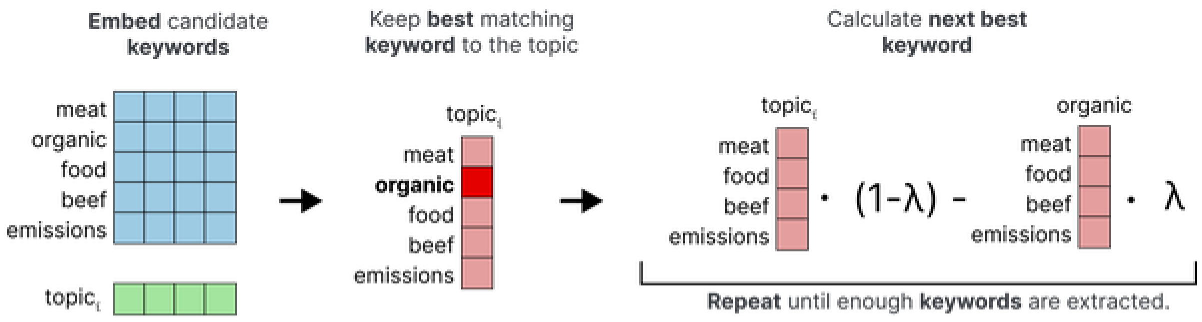


Figure 3-20. The procedure of the Maximal Marginal Relevance representation model. The diversity of the resulting keywords is represented by lambda (λ).

We can use an algorithm, called Maximal Marginal Relevance (MMR) to diversify our topic representations. The algorithm starts with the best matching keyword to a topic and then iteratively calculates the next best keyword whilst taking a certain degree of diversity into account. In other words, it takes a number of candidate topic keywords, for example, 30, and tries to pick the top 10 keywords that are best representative of the topic but are also diverse from one another.

```
# Maximal Marginal Relevance
from bertopic.representation import MaximalMargi
```

```
representation_model = MaximalMarginalRelevance(

# Use the representation model in BERTopic on top
topic_model.update_topics(abstracts, representati

# Show topic differences
topic_differences(topic_model, original_topics)
```

```
Topic: 0 question | qa | questions | answer | answering --> qa | questions | answering | comprehension | retrieval
```

```
Topic: 1 hate | offensive | speech | detection | toxic --> speech | abusive | toxicity | platforms | hateful
```

```
Topic: 2 summarization | summaries | summary | abstractive | extractive --> summarization | extractive | multidocument | documents | evaluation
```

```
Topic: 3 parsing | parser | dependency | amr | parsers --> amr | parsers | treebank | syntactic | constituent
```

```
Topic: 4 word | embeddings | embedding |
similarity | vectors --> embeddings | similarity |
vector | word2vec | glove
```

```
Topic: 5 gender | bias | biases | debiasing |
fairness --> gender | bias | fairness |
stereotypes | embeddings
```

```
Topic: 6 relation | extraction | re | relations |
entity --> extraction | relations | entity |
documentlevel | docre
```

```
Topic: 7 prompt | fewshot | prompts | incontext |
tuning --> prompts | zeroshot | plms |
metalearning | label
```

```
Topic: 8 aspect | sentiment | absa | aspectbased |
opinion --> sentiment | absa | aspects |
extraction | polarities
```

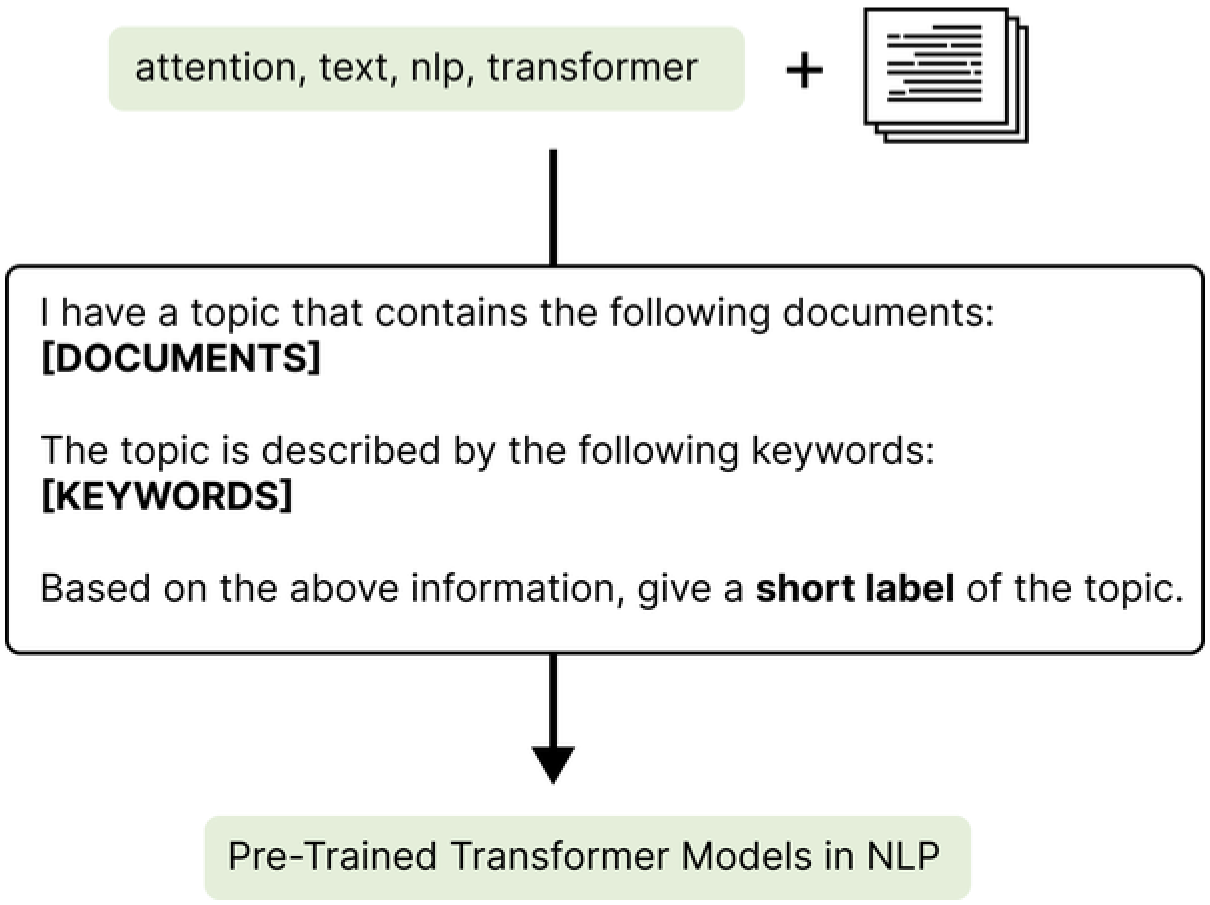
```
Topic: 9 explanations | explanation | rationales |
rationale | interpretability --> explanations |
interpretability | saliency | faithfulness |
methods
```

The resulting topics are much more diverse! Topic **XXX**, which originally used a lot of “summarization” words, the topic only contains the word “summarization”. Also, duplicates, like “embedding” and “embeddings” are now removed.

Text Generation

Text generation models have shown great potential in 2023. They perform well across a wide range of tasks and allow for extensive creativity in prompting. Their capabilities are not to be underestimated and not using them in BERTopic would frankly be a waste. We talked at length about these models in Chapter **XXX**, but it’s useful now to see how they tie into the topic modeling process.

As illustrated in Figure 3-21, we can use them in BERTopic efficiently by focusing on generating output on a topic level and not a document level. This can reduce the number of API calls from millions (e.g., millions of abstracts) to a couple of hundred (e.g., hundreds of topics). Not only does this significantly speed up the generation of topic labels, but you also do not need a massive amount of credits when using an external API, such as Cohere or OpenAI.


attention, text, nlp, transformer
+
Pre-Trained Transformer Models in NLP

I have a topic that contains the following documents:
[DOCUMENTS]

The topic is described by the following keywords:
[KEYWORDS]

Based on the above information, give a **short label** of the topic.

Figure 3-21. Use text generative LLMs and prompt engineering to create labels for topics from keywords and documents related to each topic.

Prompting

As was illustrated back in Figure 3-21, one major component of text generation is prompting. In BERTopic this is just as important since we want to give enough information to the model such that it can decide what the topic is about. Prompts in BERTopic generally look something like this:

```
prompt = """
I have a topic that contains the following docum
```

```
The topic is described by the following keywords

Based on the above information, give a short labe
"""
```

There are three components to this prompt. First, it mentions a few documents of a topic that best describes it. These documents are selected by calculating their c-TF-IDF representations and comparing them with the topic c-TF-IDF representation. The top 4 most similar documents are then extracted and referenced using the “**[DOCUMENTS]**” tag.

```
I have a topic that contains the following docume
```

Second, the keywords that make up a topic are also passed to the prompt and referenced using the “**[KEYWORDS]**” tag. These keywords could also already be optimized using KeyBERTInspired, PartOfSpeech, or any representation model.

```
The topic is described by the following keywords
```

Third, we give specific instructions to the Large Language Model. This is just as important as the steps before since this will decide how the model generates the label.

```
Based on the above information, give a short labe
```

The prompt will be rendered as follows for topic XXX:

```
"""
I have a topic that contains the following docume

- Our videos are also made possible by your suppo
- If you want to help us make more videos, you ca
- If you want to help us make more videos, you ca
- And if you want to support us in our endeavor t

The topic is described by the following keywords

Based on the above information, give a short labe
"""
```

HuggingFace

Fortunately, as with most Large Language Models, there is an enormous amount of open-source models that we can use through HuggingFace's Modelhub.

One of the most well-known open-source Large Language Models that is optimized for text generation, is one from the Flan-T5 family of generation models. What is interesting about these models is that they have been trained using a method called **instruction tuning**. By fine-tuning T5 models on many tasks phrased as instructions, the model learns to follow specific instructions and tasks.

BERTopic allows for using such a model to generate topic labels. We create a prompt and ask it to create topics based on the keywords of each topic, labeled with the `[KEYWORDS]` tag.

```
from transformers import pipeline
from bertopic.representation import TextGenerati

# Text2Text Generation with Flan-T5
generator = pipeline('text2text-generation', mod
representation_model = TextGeneration(generator)

# Use the representation model in BERTopic on to
topic_model.update_topics(abstracts, representat

# Show topic differences
topic_differences(topic_model, original_topics)
```

```
Topic: 0 speech | asr | recognition | acoustic |
endtoend --> audio grammatical recognition
```

```
Topic: 1 clinical | medical | biomedical | notes |
health --> ehr
```

```
Topic: 2 summarization | summaries | summary |
abstractive | extractive --> mds
```

```
Topic: 3 parsing | parser | dependency | amr |
parsers --> parser
```

```
Topic: 4 hate | offensive | speech | detection |
toxic --> Twitter
```

```
Topic: 5 word | embeddings | embedding | vectors |
similarity --> word2vec
```

```
Topic: 6 gender | bias | biases | debiasing |
fairness --> gender bias
```

```
Topic: 7 ner | named | entity | recognition |
nested --> ner
```

```
Topic: 8 prompt | fewshot | prompts | incontext |
tuning --> gpt3
```

```
Topic: 9 relation | extraction | re | relations |
distant --> docre
```

There are interesting topic labels that are created but we can also see that the model is not perfect by any means.

OpenAI

When we are talking about generative AI, we cannot forget about ChatGPT and its incredible performance. Although not open source, it makes for an interesting model that has changed the AI field in just a few months. We can select any text generation model from OpenAI's collection to use in BERTopic.

As this model is trained on RLHF and optimized for chat purposes, prompting is quite satisfying with this model.

```
from bertopic.representation import OpenAI

# OpenAI Representation Model
prompt = """
I have a topic that contains the following docum
The topic is described by the following keywords

Based on the information above, extract a short 
topic: <topic label>
"""
```

```
representation_model = OpenAI(model="gpt-3.5-tur

# Use the representation model in BERTopic on to
topic_model.update_topics(abstracts, representat

# Show topic differences
topic_differences(topic_model, original_topics)
```

```
Topic: 0 speech | asr | recognition | acoustic |
endtoend --> audio grammatical recognition
```

```
Topic: 1 clinical | medical | biomedical | notes |
health --> ehr
```

```
Topic: 2 summarization | summaries | summary |
abstractive | extractive --> mds
```

```
Topic: 3 parsing | parser | dependency | amr |
parsers --> parser
```

```
Topic: 4 hate | offensive | speech | detection |
toxic --> Twitter
```

```
Topic: 5 word | embeddings | embedding | vectors |
similarity --> word2vec
```

```
Topic: 6 gender | bias | biases | debiasing | fairness --> gender bias
```

```
Topic: 7 ner | named | entity | recognition | nested --> ner
```

```
Topic: 8 prompt | fewshot | prompts | incontext | tuning --> gpt3
```

```
Topic: 9 relation | extraction | re | relations | distant --> docre
```

Since we expect ChatGPT to return the topic in a specific format, namely “topic: <topic label>” it is important to instruct the model to return it as such when we create a custom prompt. Note that we also add the `delay_in_seconds` parameter to create a constant delay between API calls in case you have a free account.

Cohere

As with OpenAI, we can use Cohere’s API within BERTopic on top of its pipeline to further fine-tune the topic representations with a generative text model. Make sure to grab an API key and you can start generating topic representations.

```
import cohere
from bertopic.representation import Cohere

# Cohere Representation Model
co = cohere.Client(my_api_key)
representation_model = Cohere(co)

# Use the representation model in BERTopic on top
topic_model.update_topics(abstracts, representat

# Show topic differences
topic_differences(topic_model, original_topics)
```

```
Topic: 0 speech | asr | recognition | acoustic |
endtoend --> audio grammatical recognition
```

```
Topic: 1 clinical | medical | biomedical | notes |
health --> ehr
```

```
Topic: 2 summarization | summaries | summary |
abstractive | extractive --> mds
```

```
Topic: 3 parsing | parser | dependency | amr |
parsers --> parser
```

```
Topic: 4 hate | offensive | speech | detection |
toxic --> Twitter
```

```
Topic: 5 word | embeddings | embedding | vectors |
similarity --> word2vec
```

```
Topic: 6 gender | bias | biases | debiasing |
fairness --> gender bias
```

```
Topic: 7 ner | named | entity | recognition |
nested --> ner
```

```
Topic: 8 prompt | fewshot | prompts | incontext |
tuning --> gpt3
```

```
Topic: 9 relation | extraction | re | relations |
distant --> docre
```

LangChain

To take things a step further with Large Language Models, we can leverage the LangChain framework. It allows for any of the previous text generation methods to be supplemented with additional information or even chained together. Most notably, LangChain connects language models to other sources of data to enable them to interact with their environment.

For example, we could use it to build a vector database with OpenAI and apply ChatGPT on top of that database. As we want to minimize the amount of information LangChain needs, the most representative documents are passed to the package. Then, we could use any LangChain-supported language model to extract the topics. The example below demonstrates the use of OpenAI with LangChain.

```
from langchain.llms import OpenAI
from langchain.chains.question_answering import l
from bertopic.representation import LangChain

# Langchain representation model
chain = load_qa_chain(OpenAI(temperature=0, opena
representation_model = LangChain(chain)

# Use the representation model in BERTopic on top
topic_model.update_topics(abstracts, representat
```

```
# Show topic differences
topic_differences(topic_model, original_topics)
```

`Topic: 0 speech | asr | recognition | acoustic | endtoend --> audio grammatical recognition`

`Topic: 1 clinical | medical | biomedical | notes | health --> ehr`

```
Topic: 2 summarization | summaries | summary | abstractive | extractive --> mds
```

```
Topic: 3 parsing | parser | dependency | amr | parsers --> parser
```

```
Topic: 4 hate | offensive | speech | detection | toxic --> Twitter
```

```
Topic: 5 word | embeddings | embedding | vectors | similarity --> word2vec
```

```
Topic: 6 gender | bias | biases | debiasing | fairness --> gender bias
```

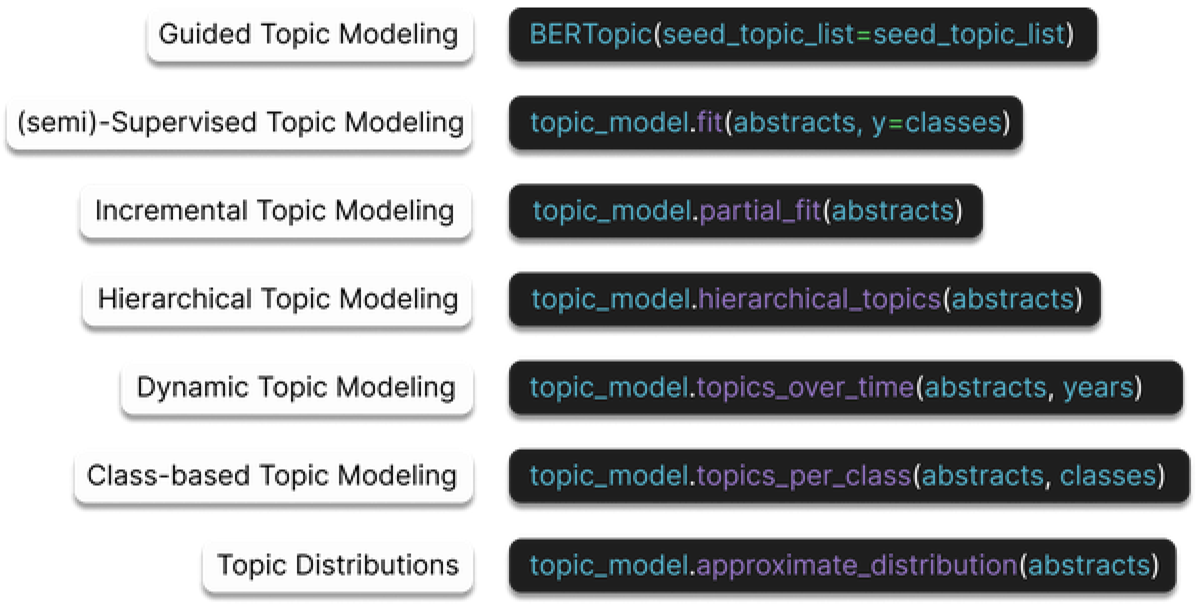
```
Topic: 7 ner | named | entity | recognition | nested --> ner
```

```
Topic: 8 prompt | fewshot | prompts | incontext | tuning --> gpt3
```

```
Topic: 9 relation | extraction | re | relations | distant --> docre
```

Topic Modeling Variations

The field of topic modeling is quite broad and ranges from many different applications to variations of the same model. This also holds for BERTopic as it has implemented a wide range of variations for different purposes, such as dynamic, (semi-) supervised, online, hierarchical, and guided topic modeling. Figure 3-22-X shows a number of topic modeling variations and how to implement them in BERTopic.


Guided Topic Modeling
(semi)-Supervised Topic Modeling
Incremental Topic Modeling
Hierarchical Topic Modeling
Dynamic Topic Modeling
Class-based Topic Modeling
Topic Distributions

```
BERTopic(seed_topic_list=seed_topic_list)
topic_model.fit(abstracts, y=classes)
topic_model.partial_fit(abstracts)
topic_model.hierarchical_topics(abstracts)
topic_model.topics_over_time(abstracts, years)
topic_model.topics_per_class(abstracts, classes)
topic_model.approximate_distribution(abstracts)
```

Figure 3-22. -X Topic Modeling Variations in BERTopic

Summary

In this chapter we discussed a cluster-based method for topic modeling, BERTopic. By leveraging a modular structure, we used a variety of Large Language Models to create document representations and fine-tune topic representations. We extracted the topics found in ArXiv abstracts and saw how we could use BERTopic’s modular structure to develop different kinds of topic representations.