[**Advanced Topic Modeling with BERTopic**](https://www.pinecone.io/learn/bertopic)

**James Briggs -** Developer Advocate

90% of the world’s data is unstructured. It is built by humans, for humans. That’s great for human consumption, but it is *very* hard to organize when we begin dealing with the massive amounts of data abundant in today’s information age.

Organization is complicated because unstructured text data is not intended to be understood by machines, and having humans process this abundance of data is wildly expensive and *very slow*.

Fortunately, there is light at the end of the tunnel. More and more of this unstructured text is becoming accessible and understood by machines. We can now [search text based on](https://www.pinecone.io/learn/dense-vector-embeddings-nlp/) [*meaning*](https://www.pinecone.io/learn/dense-vector-embeddings-nlp/), identify the sentiment of text, extract entities, and much more.

[Transformers](https://www.pinecone.io/learn/transformers/) are behind much of this. These transformers are (unfortunately) not Michael Bay’s Autobots and Decepticons and (fortunately) not buzzing electrical boxes. Our NLP transformers lie somewhere in the middle, they’re not sentient Autobots (yet), but they can understand language in a way that existed only in sci-fi until a short few years ago.

Machines with a human-like comprehension of language are pretty helpful for organizing masses of unstructured text data. In machine learning, we refer to this task as *topic modeling*, the automatic clustering of data into particular topics.

BERTopic takes advantage of the superior language capabilities of these (not yet sentient) transformer models and uses some other ML magic like UMAP and HDBSCAN (more on these later) to produce what is one of the most advanced techniques in language topic modeling today.

**BERTopic at a Glance**

We will dive into the details behind BERTopic [1], but before we do, let us see how we can use it and take a first glance at its components.

To begin, we need a dataset. We can download the dataset from HuggingFace datasets with:

from datasets import load\_dataset

data = load\_dataset('jamescalam/python-reddit')

The dataset contains data extracted using the Reddit API from the */r/python* subreddit. The code used for this (and all other examples) can [be found here](https://github.com/pinecone-io/examples/tree/master/learn/algos-and-libraries/bertopic).

Reddit thread contents are found in the **selftext** feature. Some are empty or short, so we remove them with:

data = data.filter(

lambda x: True if len(x['selftext']) > 30 else 0

)

We perform topic modeling using the **BERTopic** library. The *“basic”* approach requires just a few lines of code.

from bertopic import BERTopic

from sklearn.feature\_extraction.text import CountVectorizer

*# we add this to remove stopwords*

vectorizer\_model = CountVectorizer(ngram\_range=(1, 2), stop\_words="english")

model = BERTopic(

vectorizer\_model=vectorizer\_model,

language='english', calculate\_probabilities=True,

verbose=True

)

topics, probs = model.fit\_transform(text)

From **model.fit\_transform** we return two lists:

* **topics** contains a one-to-one mapping of inputs to their modeled *topic* (or cluster).
* **probs** contains a list of probabilities that an input belongs to their assigned topic.

We can then view the topics using **get\_topic\_info**.

In[5]:

freq = model.get\_topic\_info()

freq.head(10)

Out[5]:

Topic Count Name

0 -1 196 -1\_python\_code\_data\_using

1 0 68 0\_image\_ampx200b\_code\_images

2 1 58 1\_python\_learning\_programming\_just

3 2 44 2\_python\_django\_flask\_library

4 3 32 3\_link\_title\_thumbnail\_datepublished

5 4 28 4\_package\_python\_like\_slap

6 5 27 5\_spectra\_space\_asteroid\_training

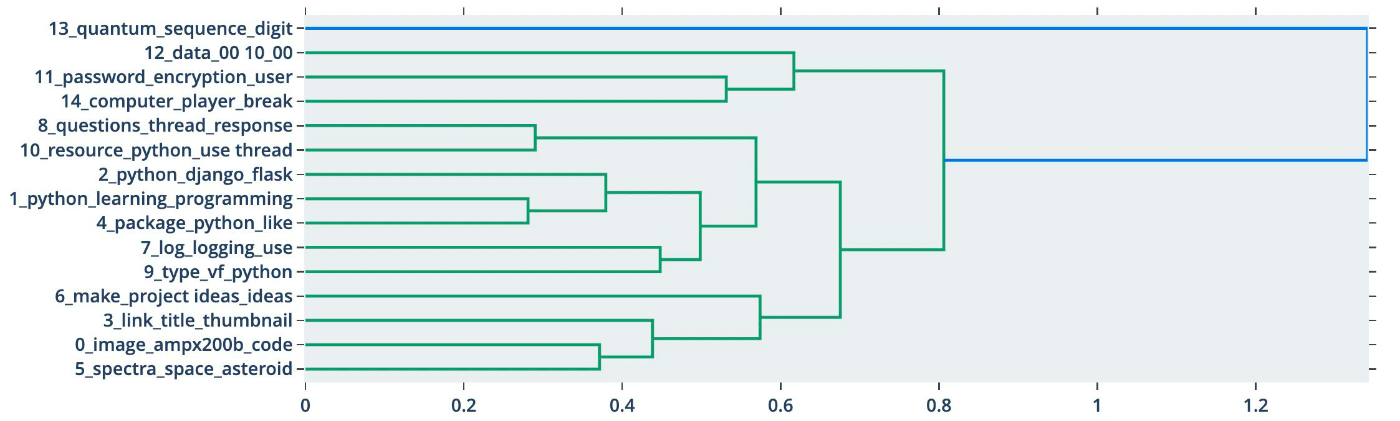
7 6 26 6\_make\_project ideas\_ideas\_comment

8 7 23 7\_log\_logging\_use\_conn

9 8 21 8\_questions\_thread\_response\_python

The top **-1** topic is typically assumed to be irrelevant, and it usually contains stop words like *“the”*, *“a”*, and *“and”*. However, we removed stop words via the **vectorizer\_model** argument, and so it shows us the *“most generic”* of topics like *“Python”*, *“code”*, and *“data”*.

The library has several built-in visualization methods like **visualize\_topics**, **visualize\_hierarchy**, and **visualize\_barchart**.



BERTopic’s visualize\_hierarchy visualization allows us to view the “hierarchy” of topics.

These represent the surface level of the BERTopic library, which has excellent documentation, so we will not rehash that here. Instead, let’s try and understand *how* BERTopic works.

**Overview**

There are *four* key components used in BERTopic [2], those are:

* A transformer embedding model
* UMAP dimensionality reduction
* HDBSCAN clustering
* Cluster tagging using c-TF-IDF

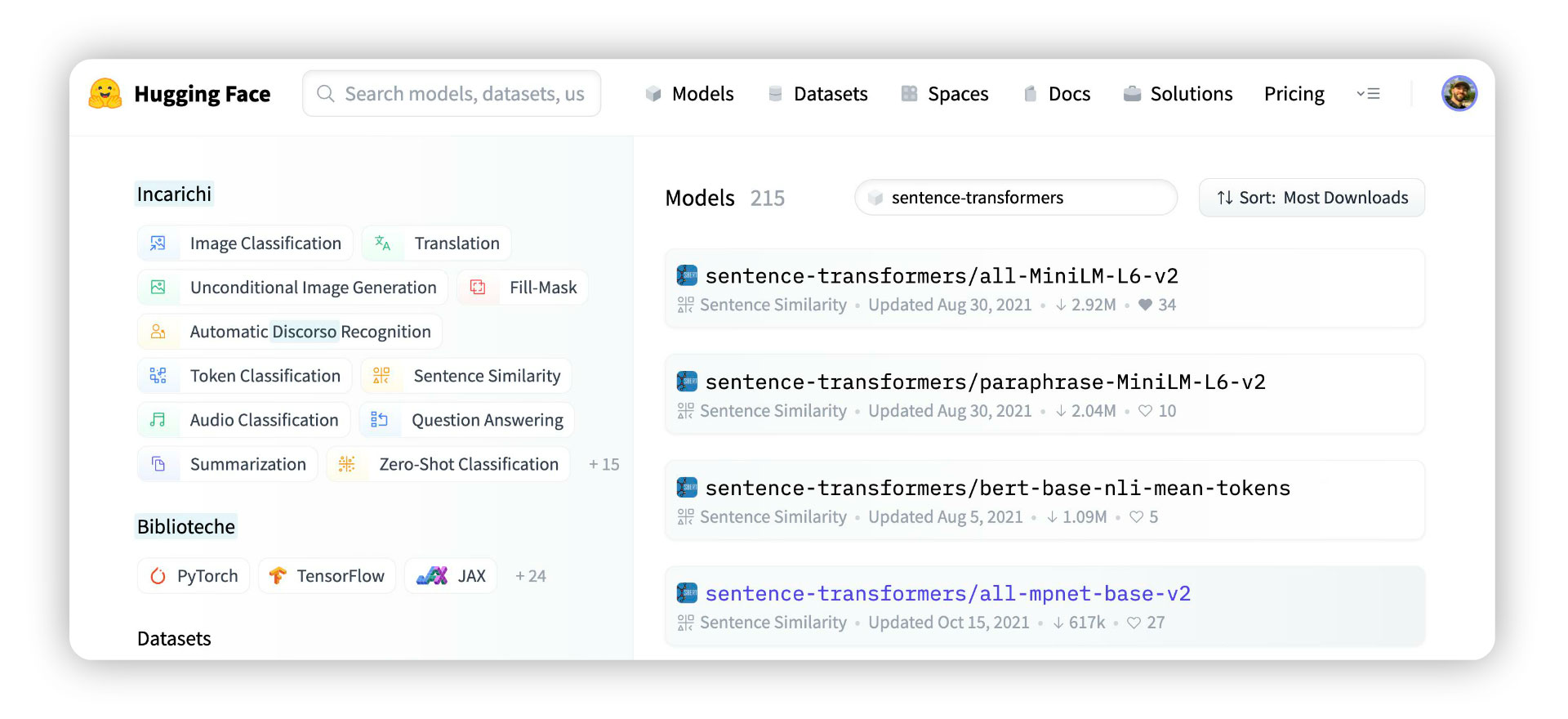
We already did *all* of this in those few lines of BERTopic code; everything is just abstracted away. However, we can optimize the process by understanding the essentials of each component. This section will work through each component *without* BERTopic, and learn how they work before returning to BERTopic at the end.

**Transformer Embedding**

BERTopic supports several libraries for encoding our text to dense vector embeddings. If we build poor quality embeddings, nothing we do in the other steps will be able to help us, so it is *very important* that we choose a suitable embedding model from one of the supported libraries, which include:

* Sentence Transformers
* Flair
* SpaCy
* Gensim
* USE (from TF Hub)

Of the above, the *Sentence Transformers* library provides the most extensive library of high-performing [sentence embedding models](https://www.pinecone.io/learn/sentence-embeddings/). They can be found on HuggingFace Hub by searching for *“sentence-transformers”*.



We can find official sentence transformer models by searching for “sentence-transformers” on HuggingFace Hub.

The first result of this search is **sentence-transformers/all-MiniLM-L6-v2**, this is a popular high-performing model that creates *384*-dimensional sentence embeddings.

To initialize the model and encode our Reddit topics data, we first **pip install sentence-transformers** and then write:

In[5]:

from sentence\_transformers import SentenceTransformer

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

model

Out[5]:

SentenceTransformer(

(0): Transformer({'max\_seq\_length': 256, 'do\_lower\_case': False}) with Transformer model: BertModel

(1): Pooling({'word\_embedding\_dimension': 384, 'pooling\_mode\_cls\_token': False, 'pooling\_mode\_mean\_tokens': True, 'pooling\_mode\_max\_tokens': False, 'pooling\_mode\_mean\_sqrt\_len\_tokens': False})

(2): Normalize()

)

In[6]:

import numpy as np

from tqdm.auto import tqdm

batch\_size = 16

embeds = np.zeros((n, model.get\_sentence\_embedding\_dimension()))

for i in tqdm(range(0, n, batch\_size)):

i\_end = min(i+batch\_size, n)

batch = data['selftext'][i:i\_end]

batch\_embed = model.encode(batch)

embeds[i:i\_end,:] = batch\_embed

Out[6]:

100%|██████████| 195/195 [08:51<00:00, 2.73s/it]

Here we have encoded our text in batches of **16**. Each batch is added to the **embeds** array. Once we have all of the [sentence embeddings](https://www.pinecone.io/learn/sentence-embeddings/) in **embeds** we’re ready to move on to the next step.

**Dimensionality Reduction**

After building our embeddings, BERTopic compresses them into a lower-dimensional space. This means that our 384-dimensional vectors are transformed into two/three-dimensional vectors.

We can do this because 384 dimensions are *a lot*, and it is unlikely that we really need that many dimensions to represent our text [4]. Instead, we attempt to *compress* that information into two or three dimensions.

We do this so that the following HDBSCAN clustering step can be done more efficiently. Performing the clustering step with 384-dimensions would be desperately slow [5].

Another benefit is that we can visualize our data; this is incredibly helpful when assessing whether our data can be clustered. Visualization also helps when tuning the dimensionality reduction parameters.

To help us understand dimensionality reduction, we will start with a 3D representation of the world. You can find [the code for this part here](https://github.com/pinecone-io/examples/tree/master/learn/algos-and-libraries/bertopic).

We can apply many dimensionality reduction techniques to this data; two of the most popular choices are PCA and t-SNE.

PCA works by preserving *larger distances* (using mean squared error). The result is that the *global structure* of data is usually preserved [6]. We can see that behavior above as each continent is grouped with its neighboring continent(s). When we have easily distinguishable clusters in datasets, this can be good, but it performs poorly for more nuanced data where *local structures* are important.

t-SNE is the opposite; it preserves *local structures* rather than *global*. This localized focus results from t-SNE building a graph, connecting all of the nearest points. These local structures can indirectly suggest the global structure, but they are not strongly captured.

PCA focuses on preserving *dissimilarity* whereas t-SNE focuses on preserving *similarity*.

Fortunately, we can capture the best of both using a lesser-known technique called **U**niform **M**anifold **A**pproximation and **P**roduction (UMAP).

We can apply UMAP in Python using the UMAP library, installed using **pip install umap-learn**. To map to a 3D or 2D space using the default UMAP parameters, all we write is:

import umap

fit = umap.UMAP(n\_components=3) *# by default this is 2*

u = fit.fit\_transform(data)

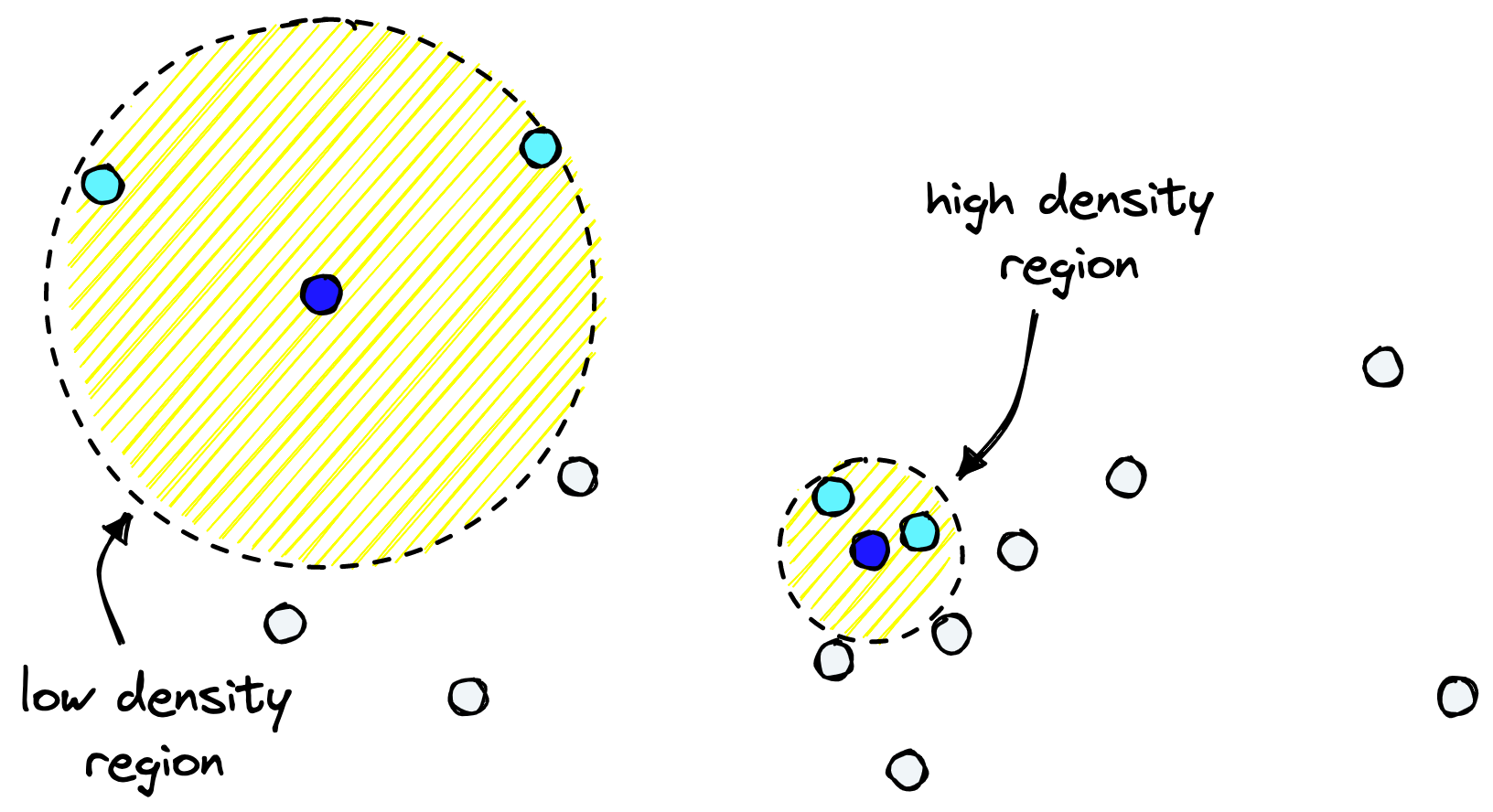
The UMAP algorithm can be fine-tuned using several parameters. Still, the simplest and most effective tuning can be achieved with just the **n\_neighbors** parameter.

For each datapoint, UMAP searches through other points and identifies the **k**th nearest neighbors [3]. It is **k**, controlled by the **n\_neighbors** parameter.

k and n\_neighbors are synonymous here. As we increase n\_neighbors the graph built by UMAP can consider more distant points and better represent the global structure.

As we increase **n\_neighbors** the graph built by UMAP can consider more distant points and better represent the global structure.

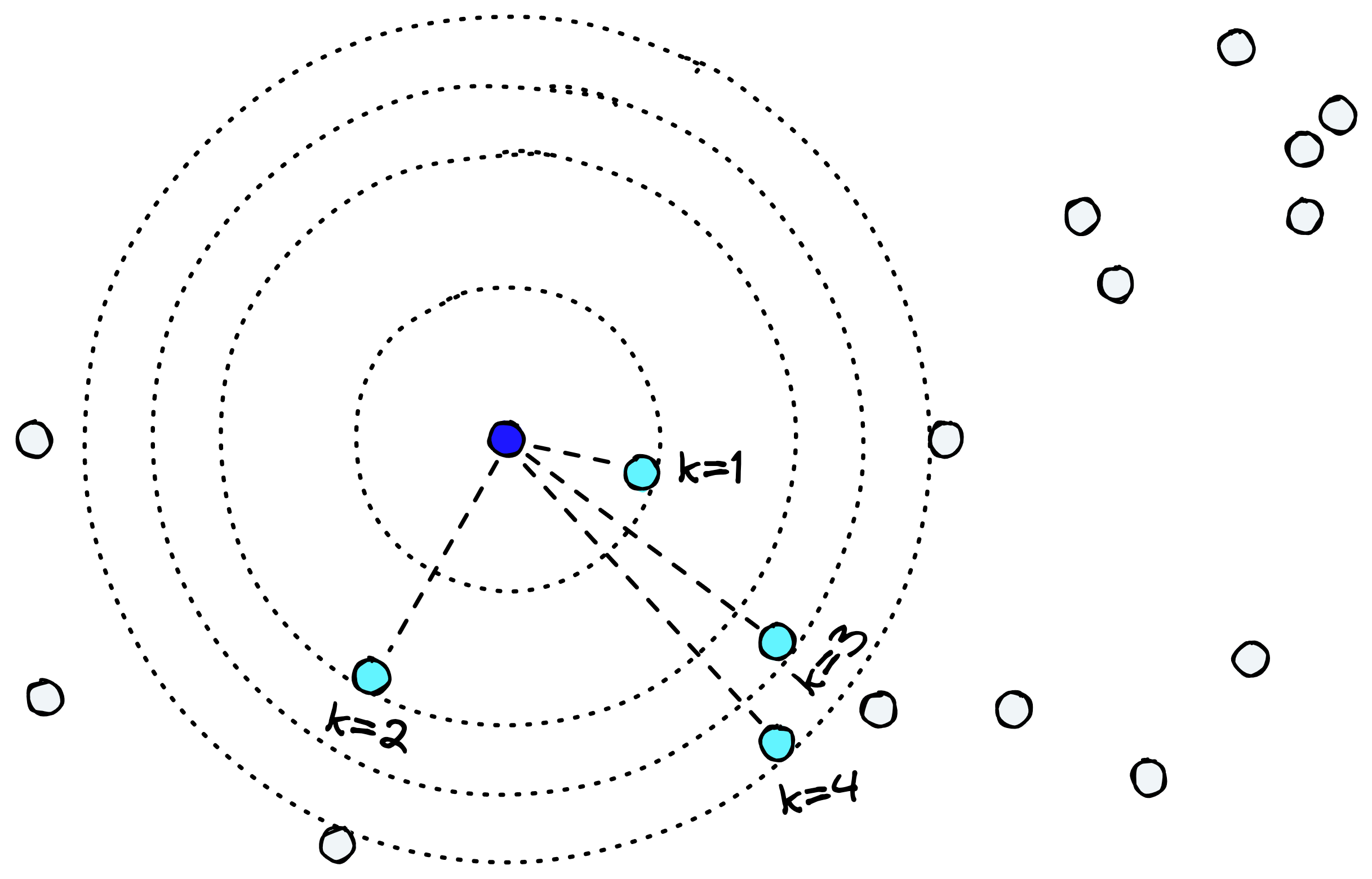
Where we have many points (high-density regions), the distance between our point and its **k**th nearest neighbor is usually smaller. In low-density regions with fewer points, the distance will be much greater.



Density is measured indirectly using the distances between kth nearest neighbors in different regions.

UMAP will attempt to preserve distances to the **k**th nearest point is what UMAP attempts to preserve when shifting to a lower dimension.

By increasing **n\_neighbors** we can preserve more global structures, whereas a lower **n\_neighbors** better preserves local structures.



Higher n\_neighbors (k) means we preserve larger distances and thus maintain more of the global structure.

Compared to other dimensionality reduction techniques like PCA or t-SNE, finding a good **n\_neighbours** value allows us to preserve *both* local and global structures relatively well.

Applying it to our 3D globe, we can see neighboring countries remain neighbors. At the same time, continents are placed correctly (with North-South inverted), and islands are separated from continents. We even have what seems to be the Spanish Peninsula in “western Europe”.

UMAP maintains distinguishable features that are not preserved by PCA and a better global structure than t-SNE. This is a great overall example of where the benefit of UMAP lies.

UMAP can also be used as a supervised dimensionality reduction method by passing labels to the **target** argument if we have labeled data. It is possible to produce even more meaningful structures using this supervised approach.

With all that in mind, let us apply UMAP to our Reddit topics data. Using **n\_neighbors** of **3**-**5** seems to work best. We can add **min\_dist=0.05** to allow UMAP to place points closer together (the default value is **1.0**); this helps us separate the three *similar* topics from *r/Python*, *r/LanguageTechnology*, and *r/pytorch*.

fit = umap.UMAP(n\_neighbors=3, n\_components=3, min\_dist=0.05)

u = fit.fit\_transform(embeds)

With our data reduced to a lower-dimensional space and topics easily visually identifiable, we’re in an excellent spot to move on to clustering.

**HDBSCAN Clustering**

We have visualized the UMAP reduced data using the existing *sub* feature to color our clusters. It looks pretty, but we don’t usually perform topic modeling to label already labeled data. If we assume that we have no existing labels, our UMAP visual will look like this:

Share via: