Open in app ↗

Medium Search Write

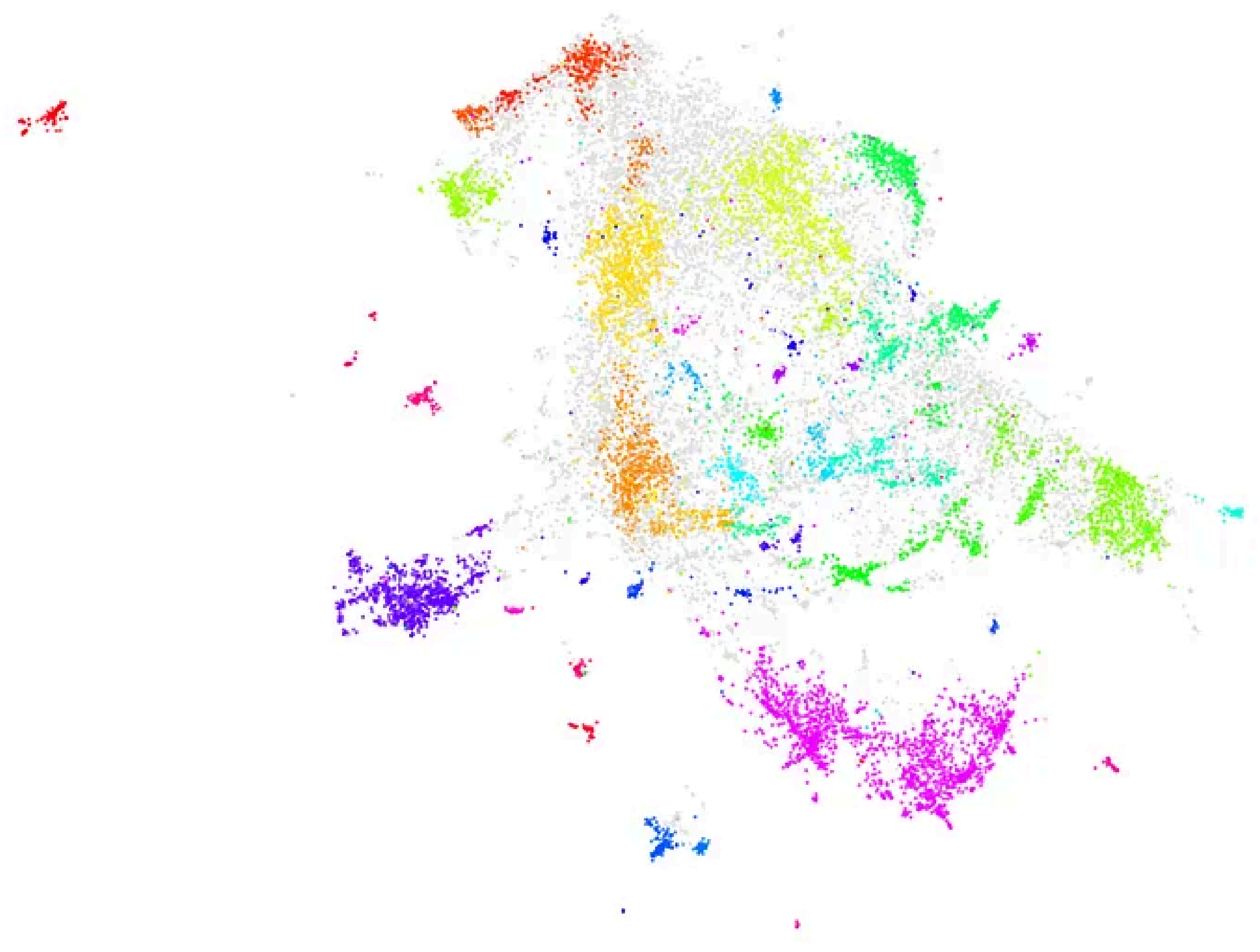


Image by the author.

✦ Member-only story

Topic Modeling with BERT

Leveraging BERT and TF-IDF to create easily interpretable topics.

Maarten Grootendorst · Follow

Published in Towards Data Science · 8 min read · Oct 5, 2020

3K 25

Often when I am approached by a product owner to do some NLP-based analyses, I am typically asked the following question:

> ‘Which topic can frequently be found in these documents?’

Void of any categories or labels I am forced to look into unsupervised techniques to extract these topics, namely **Topic Modeling.**

Although topic models such as LDA and NMF have shown to be good starting points, I always felt it took quite some effort through hyperparameter tuning to create meaningful topics.

Moreover, I wanted to use transformer-based models such as **BERT** as they have shown amazing results in various NLP tasks over the last few years. **Pre-trained models** are especially helpful as they are supposed to contain more accurate representations of words and sentences.

A few weeks ago I saw this great project named Top2Vec* which leveraged document- and word embeddings to create topics that were easily interpretable. I started looking at the code to generalize Top2Vec such that it could be used with pre-trained transformer models.

The great advantage of Doc2Vec is that the resulting document- and word embeddings are jointly embedding in the same space which allows document embeddings to be represented by nearby word embeddings.

Unfortunately, this proved to be difficult as BERT embeddings are token-based and do not necessarily occupy the same space**.

Instead, I decided to come up with a different algorithm that could use BERT and 🤗 transformers embeddings. The result is BERTopic, an algorithm for generating topics using state-of-the-art embeddings.

The main topic of this article will not be the use of BERTopic but a **tutorial** on how to use BERT to create your own **topic model**.

PAPER*: Angelov, D. (2020). Top2Vec: Distributed Representations of Topics. *arXiv preprint arXiv:2008.09470*.

NOTE**: Although you could have them occupy the same space, the resulting size of the word embeddings is quite large due to the contextual nature of BERT. Moreover, there is a chance that the resulting sentence- or document embeddings will degrade in quality.

1. Data & Packages

For this example, we use the famous `20 Newsgroups` dataset which contains roughly 18000 newsgroups posts on 20 topics. Using Scikit-Learn, we can quickly download and prepare the data:

```python
from sklearn.datasets import fetch_20newsgroups
data = fetch_20newsgroups(subset='all')['data']
```

newsgroups.py hosted with ❤️ by GitHub view raw

If you want to speed up training, you can select the subset `train` as it will decrease the number of posts you extract.

NOTE: If you want to apply topic modeling not on the entire document but on the paragraph level, I would suggest splitting your data before creating the embeddings.

2. Embeddings

The very first step we have to do is converting the documents to numerical data. We use **BERT** for this purpose as it extracts different embeddings based on the context of the word. Not only that, there are many pre-trained models available ready to be used.

How you generate the BERT embeddings for a document is up to you. However, I prefer to use the `sentence-transformers` package as the resulting embeddings have shown to be of high quality and typically work quite well for document-level embeddings.

Install the package with `pip install sentence-transformers` before generating the document embeddings. If you run into issues installing this package, then it is worth installing Pytorch first.

Then, run the following code to transform your documents in 512-dimensional vectors:

```python
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('distilbert-base-nli-mean-tokens')
embeddings = model.encode(data, show_progress_bar=True)
```

sentence.py hosted with ❤ by GitHub view raw

We are using **Distilbert** as it gives a nice balance between speed and performance. The package has several multi-lingual models available for you to use.

NOTE: Since transformer models have a token limit, you might run into some errors when inputting large documents. In that case, you could consider splitting documents into paragraphs.

3. Clustering

We want to make sure that documents with similar topics are clustered together such that we can find the topics within these clusters. Before doing so, we first need to lower the dimensionality of the embeddings as many clustering algorithms handle high dimensionality poorly.

UMAP

Out of the few dimensionality reduction algorithms, UMAP is arguably the best performing as it keeps a significant portion of the high-dimensional local structure in lower dimensionality.

Install the package with `pip install umap-learn` before we lower the dimensionality of the document embeddings. We reduce the dimensionality to 5 while keeping the size of the local neighborhood at 15. You can play around with these values to optimize for your topic creation. Note that a too low dimensionality results in a loss of information while a too high dimensionality results in poorer clustering results.

```python
import umap
umap_embeddings = umap.UMAP(n_neighbors=15,
                            n_components=5,
                            metric='cosine').fit_transform(embeddings)
```

umap.py hosted with ❤ by GitHub view raw

HDBSAN

After having reduced the dimensionality of the documents embeddings to 5, we can cluster the documents with **HDBSCAN**. HDBSCAN is a density-based algorithm that works quite well with UMAP since UMAP maintains a lot of local structure even in lower-dimensional space. Moreover, HDBSCAN does not force data points to clusters as it considers them outliers.

Install the package with `pip install hdbscan` then create the clusters:

```
import hdbscan
cluster = hdbscan.HDBSCAN(min_cluster_size=15,
                          metric='euclidean',
                          cluster_selection_method='eom').fit(umap_embeddings)
```

hdbscan.py hosted with ❤ by GitHub view raw

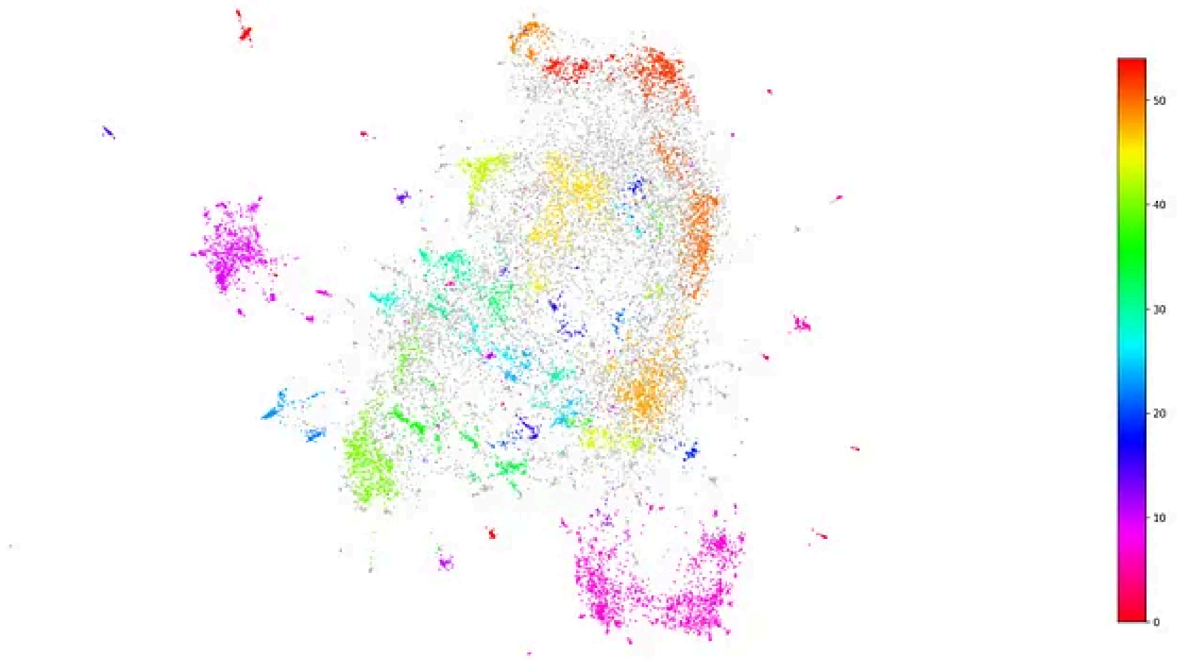
Great! We now have clustered similar documents together which should represent the topics that they consist of. To visualize the resulting clusters we can further reduce the dimensionality to 2 and visualize the outliers as grey points:

```python
import matplotlib.pyplot as plt

# Prepare data
umap_data = umap.UMAP(n_neighbors=15, n_components=2, min_dist=0.0, metric='cosine').fit
result = pd.DataFrame(umap_data, columns=['x', 'y'])
result['labels'] = cluster.labels_

# Visualize clusters
fig, ax = plt.subplots(figsize=(20, 10))
outliers = result.loc[result.labels == -1, :]
clustered = result.loc[result.labels != -1, :]
plt.scatter(outliers.x, outliers.y, color='#BDBDBD', s=0.05)
plt.scatter(clustered.x, clustered.y, c=clustered.labels, s=0.05, cmap='hsv_r')
plt.colorbar()
```

visualize_clusters.py hosted with ❤ by GitHub view raw



Topics visualized by reducing sentenced embeddings to 2-dimensional space. Image by the author.

It is difficult to visualize the individual clusters due to the number of topics generated (~55). However, we can see that even in 2-dimensional space some local structure is kept.

NOTE: You could skip the dimensionality reduction step if you use a clustering algorithm that can handle high dimensionality like a cosine-based k-Means.

4. Topic Creation

What we want to know from the clusters that we generated, is what makes one cluster, based on their content, different from another?

> *How can we derive topics from clustered documents?*

To solve this, I came up with a class-based variant of TF-IDF (**c-TF-IDF**), that would allow me to extract what makes each set of documents unique compared to the other.

The intuition behind the method is as follows. When you apply TF-IDF as usual on a set of documents, what you are basically doing is comparing the importance of words between documents.

What if, we instead treat all documents in a single category (e.g., a cluster) as **a single document** and then apply TF-IDF? The result would be a very long document per category and the resulting TF-IDF score would demonstrate the important words in a topic.

c-TF-IDF

To create this class-based TF-IDF score, we need to first create a single document for each cluster of documents:

```python
docs_df = pd.DataFrame(data, columns=["Doc"])
docs_df['Topic'] = cluster.labels_
docs_df['Doc_ID'] = range(len(docs_df))
docs_per_topic = docs_df.groupby(['Topic'], as_index = False).agg({'Doc': ' '.join})
```

doc_prep.py hosted with ❤ by **GitHub** view raw

Then, we apply the class-based TF-IDF:

$$c - TF - IDF_i = \frac{t_i}{w_i} \times \log \frac{m}{\sum_j^n t_j}$$

class-based TF-IDF by joining documents within a class. Image by the author.

Where the **frequency** of each word `t` is extracted for each class `i` and divided by the total number of words `w`. This action can be seen as a form of regularization of frequent words in the class. Next, the total, unjoined, number of documents `m` is divided by the total frequency of word `t` across all classes `n`.

```python
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer

def c_tf_idf(documents, m, ngram_range=(1, 1)):
    count = CountVectorizer(ngram_range=ngram_range, stop_words="english").fit(documents
    t = count.transform(documents).toarray()
    w = t.sum(axis=1)
    tf = np.divide(t.T, w)
    sum_t = t.sum(axis=0)
    idf = np.log(np.divide(m, sum_t)).reshape(-1, 1)
    tf_idf = np.multiply(tf, idf)

    return tf_idf, count

tf_idf, count = c_tf_idf(docs_per_topic.Doc.values, m=len(data))
```

ctfidf.py hosted with ❤ by **GitHub** view raw

Now, we have a single **importance** value for each word in a cluster which can be used to create the topic. If we take the top 10 most important words in each cluster, then we would get a good representation of a cluster, and thereby a topic.

Topic Representation

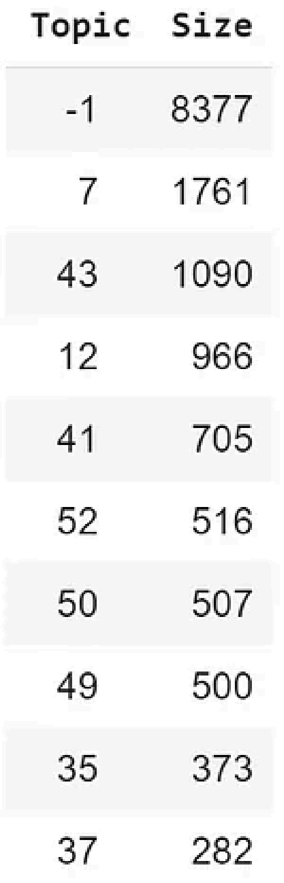
In order to create a topic representation, we take the top 20 words per topic based on their c-TF-IDF scores. The higher the score, the more representative it should be of its topic as the score is a proxy of information density.

```python
def extract_top_n_words_per_topic(tf_idf, count, docs_per_topic, n=20):
    words = count.get_feature_names()
    labels = list(docs_per_topic.Topic)
    tf_idf_transposed = tf_idf.T
    indices = tf_idf_transposed.argsort()[:, -n:]
    top_n_words = {label: [(words[j], tf_idf_transposed[i][j]) for j in indices[i]][::-1
    return top_n_words

def extract_topic_sizes(df):
    topic_sizes = (df.groupby(['Topic'])
                     .Doc
                     .count()
                     .reset_index()
                     .rename({"Topic": "Topic", "Doc": "Size"}, axis='columns')
                     .sort_values("Size", ascending=False))
    return topic_sizes

top_n_words = extract_top_n_words_per_topic(tf_idf, count, docs_per_topic, n=20)
topic_sizes = extract_topic_sizes(docs_df); topic_sizes.head(10)
```

We can use `topic_sizes` to view how frequent certain topics are:

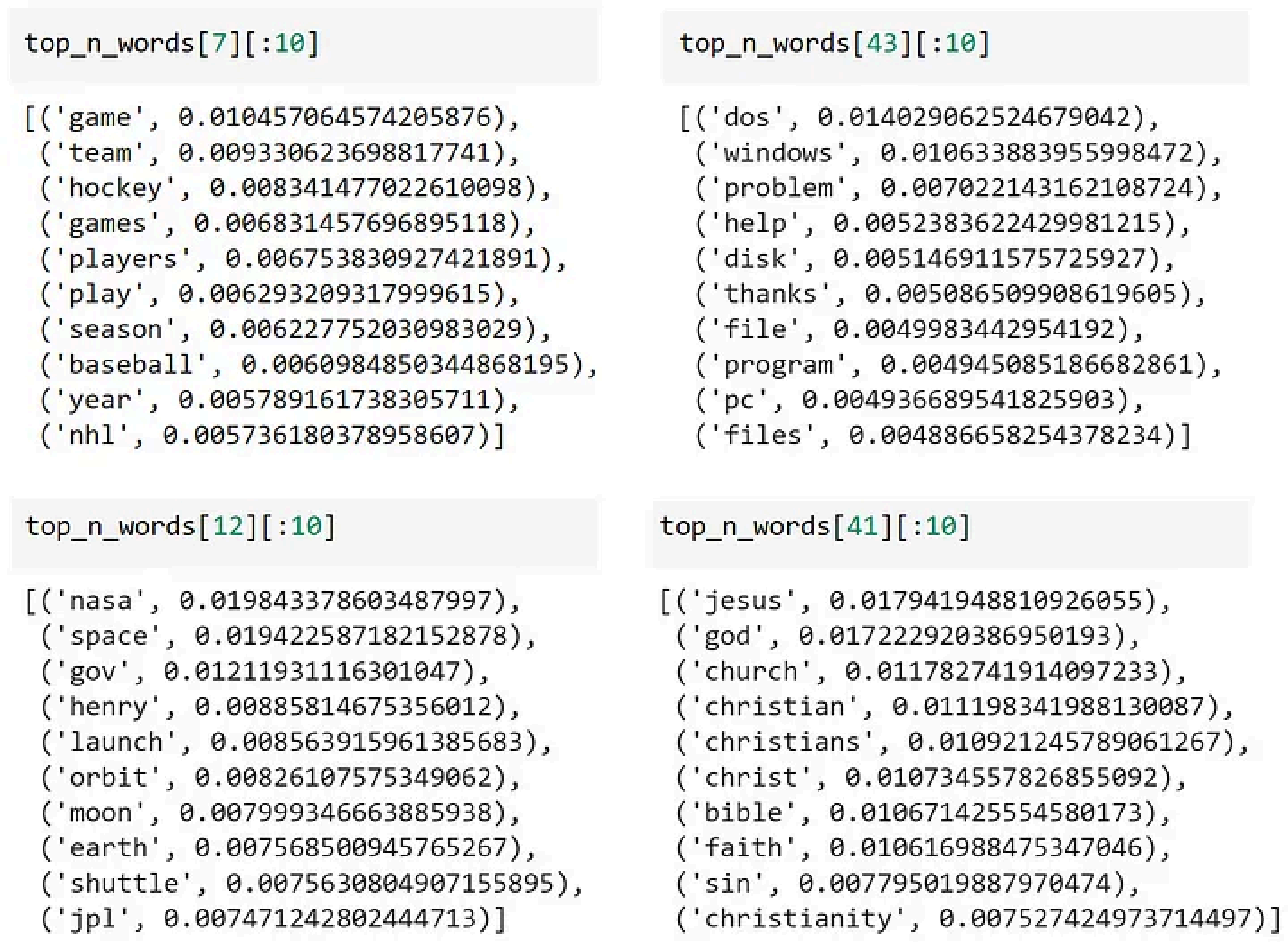


Topic	Size
-1	8377
7	1761
43	1090
12	966
41	705
52	516
50	507
49	500
35	373
37	282

Image by the author.

The topic name `-1` refers to all documents that did not have any topics assigned. The great thing about HDBSCAN is that not all documents are forced towards a certain cluster. If no cluster could be found, then it is simply an outlier.

We can see that topics 7, 43, 12, and 41 are the largest clusters that we could create. To view the words belonging to those topics, we can simply use the dictionary `top_n_words` to access these topics:



```
top_n_words[7][:10]
```

```
[('game', 0.010457064574205876),
 ('team', 0.009330623698817741),
 ('hockey', 0.008341477022610098),
 ('games', 0.006831457696895118),
 ('players', 0.006753830927421891),
 ('play', 0.006293209317999615),
 ('season', 0.006227752030983029),
 ('baseball', 0.006098485034486195),
 ('year', 0.005789161738305711),
 ('nhl', 0.005736180378958607)]
```

```
top_n_words[43][:10]
```

```
[('dos', 0.014029062524679042),
 ('windows', 0.010633883955998472),
 ('problem', 0.007022143162108724),
 ('help', 0.005238362242998121),
 ('disk', 0.005146911575725927),
 ('thanks', 0.005086509908619605),
 ('file', 0.0049983442954192),
 ('program', 0.004945085186682861),
 ('pc', 0.004936689541825903),
 ('files', 0.004886658254378234)]
```

```
top_n_words[12][:10]
```

```
[('nasa', 0.019843378603487997),
 ('space', 0.019422587182152878),
 ('gov', 0.01211931116301047),
 ('henry', 0.00885814675356012),
 ('launch', 0.008563915961385683),
 ('orbit', 0.00826107575349062),
 ('moon', 0.007999346663885938),
 ('earth', 0.007568500945765267),
 ('shuttle', 0.007563080490715589),
 ('jpl', 0.007471242802444713)]
```

```
top_n_words[41][:10]
```

```
[('jesus', 0.017941948810926055),
 ('god', 0.017222920386950193),
 ('church', 0.011782741914097233),
 ('christian', 0.011198341988130087),
 ('christians', 0.010921245789061267),
 ('christ', 0.010734557826855092),
 ('bible', 0.010671425554580173),
 ('faith', 0.010616988475347046),
 ('sin', 0.007795019887970474),
 ('christianity', 0.007527424973714497)]
```

Image by the author.

Looking at the largest four topics, I would say that these nicely seem to represent easily interpretable topics!

I can see sports, computers, space, and religion as clear topics that were extracted from the data.

5. Topic Reduction

There is a chance that, depending on the dataset, you will get hundreds of topics that were created! You can tweak the parameters of HDBSCAN such that you will get fewer topics through its `min_cluster_size` parameter but it does not allow you to specify the exact number of clusters.

A nifty trick that Top2Vec was using is the ability to reduce the number of topics by merging the topic vectors that were most similar to each other.

We can use a similar technique by **comparing** the c-TF-IDF vectors among topics, **merge** the most similar ones, and finally **re-calculate** the c-TF-IDF vectors to update the representation of our topics:

```
for i in range(20):
    # Calculate cosine similarity
    similarities = cosine_similarity(tf_idf.T)
    np.fill_diagonal(similarities, 0)

    # Extract label to merge into and from where
    topic_sizes = docs_df.groupby(['Topic']).count().sort_values("Doc", ascending=False)
    topic_to_merge = topic_sizes.iloc[-1].Topic
    topic_to_merge_into = np.argmax(similarities[topic_to_merge + 1]) - 1

    # Adjust topics
    docs_df.loc[docs_df.Topic == topic_to_merge, "Topic"] = topic_to_merge_into
    old_topics = docs_df.sort_values("Topic").Topic.unique()
    map_topics = {old_topic: index - 1 for index, old_topic in enumerate(old_topics)}
    docs_df.Topic = docs_df.Topic.map(map_topics)
    docs_per_topic = docs_df.groupby(['Topic'], as_index = False).agg({'Doc': ' '.join})

    # Calculate new topic words
    m = len(data)
    tf_idf, count = c_tf_idf(docs_per_topic.Doc.values, m)
    top_n_words = extract_top_n_words_per_topic(tf_idf, count, docs_per_topic, n=20)

topic_sizes = extract_topic_sizes(docs_df); topic_sizes.head(10)
```

topic_reduction.py hosted with ❤ by **GitHub** view raw

Above, we took the least common topic and merged it with the most similar topic. By repeating this 19 more times we reduced the number of topics from **56** to **36**!

NOTE: We can skip the re-calculation part of this pipeline to speed up the topic reduction step. However, it is more accurate to re-calculate the c-TF-IDF vectors as that would better represent the newly generated content of the topics. You can play around with this by, for example, update every n steps to both speed-up the process and still have good topic representations.

TIP: You can use the method described in this article (or simply use BERTopic) to also create sentence-level embeddings. The main advantage of this is the possibility to view the distribution of topics within a single document.

Thank you for reading!

If you are, like me, passionate about AI, Data Science, or Psychology, please feel free to add me on LinkedIn or follow me on Twitter.

All examples and code in this article can be found here:

MaartenGr/BERTopic

BERTopic is a topic modeling technique that leverages BERT embeddings and c-TF-IDF to create dense clusters allowing…

github.com

Machine Learning NLP Data Science Artificial Intelligence Editors Pick

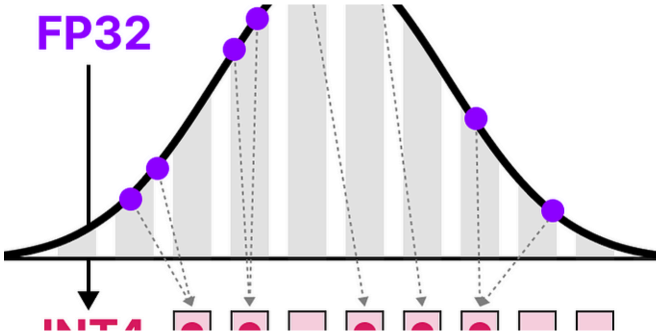
Written by Maarten Grootendorst

Follow

7K Followers · Writer for Towards Data Science

Data Scientist | Psychologist. Passionate about anything AI-related! Get in touch: www.linkedin.com/in/mgrootendorst/

More from Maarten Grootendorst and Towards Data Science



Maarten Grootendorst in Towards Data Science

A Visual Guide to Quantization

Demystifying the compression of large language models

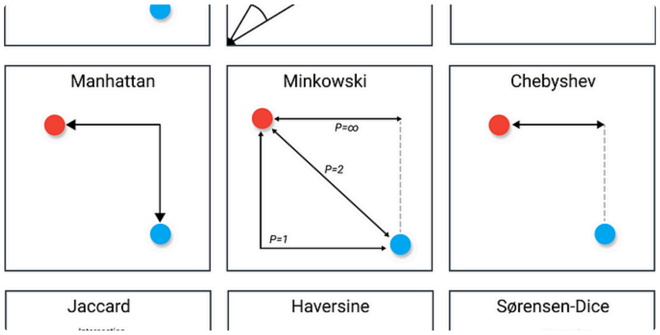
Jul 24 733 2

Shaw Talebi in Towards Data Science

5 AI Projects You Can Build This Weekend (with Python)

From beginner-friendly to advanced

Oct 8 2.9K 45



Mauro Di Pietro in Towards Data Science

GenAI with Python: Build Agents from Scratch (Complete Tutorial)

with Ollama, LangChain, LangGraph (No GPU, No APIKEY)

Sep 29 1.6K 23

Maarten Grootendorst in Towards Data Science

9 Distance Measures in Data Science

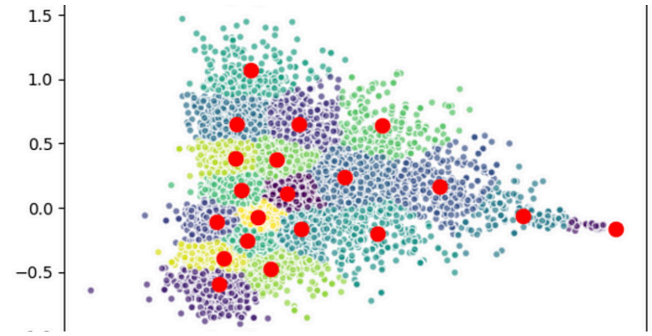
The advantages and pitfalls of common distance measures

Feb 1, 2021 4.2K 26

See all from Maarten Grootendorst

See all from Towards Data Science

Recommended from Medium



Kartheepan G

Unveiling Text Clustering: Exploring Algorithms and Text...

Introduction In today's data-driven world, the ability to effectively analyze and organize...

Apr 22 7 1

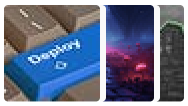
Mariya Mansurova in Towards Data Science

Topics per Class Using BERTopic

How to understand the differences in texts by categories

Sep 8, 2023 648 4

Lists



Predictive Modeling w/ Python

20 stories · 1599 saves

Natural Language Processing

1759 stories · 1358 saves

Practical Guides to Machine Learning

10 stories · 1947 saves

The New Chatbots: ChatGPT, Bard, and Beyond

12 stories · 484 saves

DhanushKumar

Topic Modelling with BERTopic

BERTopic is a topic modeling technique that leverages BERT (Bidirectional Encoder...

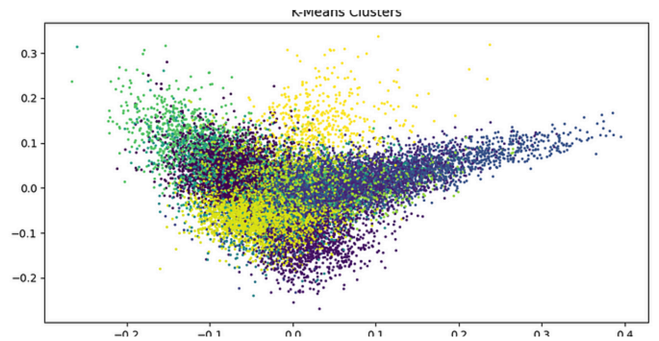
Jul 1 13

Ebad Sayed

Scikit-LLM: Scikit-Learn Meets Large Language Models

As a beginner in Python and ML, I frequently relied on scikit-learn for almost all of my...

Sep 9 70 1



Mahdi Rafati

Mastering Text Clustering with Python: A Comprehensive Guide

Clustering is a powerful technique for organizing and understanding large text...

Jun 3 11 1

Emmanuel Ikogho

Data Science is dying; here's why

Why 85% of data science projects fail

Sep 3 1.5K 72

See more recommendations