Oracle AI & Data Science Blog

Learn Data Science, Oracle AI

Clustering text documents using the natural language processing (NLP) conda pack in OCI Data Science

Wendy Yip | June 2, 2021
Data Scientist

Oracle Cloud Infrastructure (OCI) Data Science recently released two conda packs designed for natural language processing (NLP) workloads: the natural language processing conda pack for CPU and the natural language processing conda pack for GPU. Both conda packs are available to customers when they log in to OCI Data Science.

Natural language processing (NLP) refers to the area of artificial intelligence of how machines work with human language. NLP tasks include sentiment analysis, language detection, key phrase extraction, and clustering of similar documents. Our conda packs come pre-installed with many packages for NLP workloads.

Over the last few years, NLP has undergone great advances with the introduction of Transformer models such as BERT (Bidirectional Encoder Representations from Transformers). Our conda packs include Hugging Face's state-of-the-art transformers library, which contains pre-trained models for different NLP tasks. In addition, it includes the versatile libraries NLTK and Scikit-learn for pre-processing text data and model building, various wrappers around BERT like key-bert, and deep learning accelerating frameworks PyTorch Lightning building, various wrappers around BERT like key-bert, and deep learning accelerating frameworks PyTorch Lightning.

We are going to show an example of how to use the NLP conda pack to process and group a set of documents. We are going to use the 20 Newsgroups dataset, which contains ~20,000 forum posts from 20 different topics.

Load the dataset

First, we import the necessary libraries. We are going to use Scikit-learn for loading in the dataset and pre-processing the data. Also, we are going to use UMAP, which is a dimensionality reduction library, along with Matplotlib and Bokeh for data visualization.

```
import pandas as pd
import umap
import umap.plot

# Used to get the data
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer

# Some plotting libraries
import matplotlib.pyplot as plt
from bokeh.plotting import show, save, output_notebook, output_file
```

Let’s load the dataset. Scikit-learn has a built-in function for loading the 20 Newsgroups dataset.

```
dataset = fetch_20newsgroups(subset='all',
                             shuffle=True)
```

These are the 20 Newsgroups categories that a post can belong to.

```
dataset.target_names
```

```
['alt.atheism',
 'comp.graphics',
 'comp.os.ms-windows.misc',
 'comp.sys.ibm.pc.hardware',
 'comp.sys.mac.hardware',
 'comp.windows.x',
 'misc.forsale',
 'rec.autos',
 'rec.motorcycles',
 'rec.sport.baseball',
 'rec.sport.hockey',
 'sci.crypt',
 'sci.electronics',
 'sci.med',
 'sci.space',
 'soc.religion.christian',
 'talk.politics.guns',
 'talk.politics.mideast',
 'talk.politics.misc',
 'talk.religion.misc']
```

Let’s look at one post.

```
Category: rec.sport.hockey
---------------------------
From: Mamatha Devineni Ratnam <mr47+@andrew.cmu.edu>
Subject: Pens fans reactions
Organization: Post Office, Carnegie Mellon, Pittsburgh, PA
```

```
Organization: Post Office, Carnegie Mellon, Pittsburgh, PA
Lines: 12
NNTP-Posting-Host: po4.andrew.cmu.edu



I am sure some bashers of Pens fans are pretty confused about the lack
of any kind of posts about the recent Pens massacre of the Devils. Actually,
I am  bit puzzled too and a bit relieved. However, I am going to put an end
to non-PIttsburghers' relief with a bit of praise for the Pens. Man, they
are killing those Devils worse than I thought. Jagr just showed you why
he is much better than his regular season stats. He is also a lot
fo fun to watch in the playoffs. Bowman should let JAgr have a lot of
fun in the next couple of games since the Pens are going to beat the pulp out of Jersey anyway. I was very disappointed not to see the
Islanders lose the final
regular season game.          PENS RULE!!!
```

We notice metadata such as title and name. These are parts of the data that a classifier may overfit on. We can use techniques such as TF-IDF to minimize the undesirable data from polluting a classifier (We will discuss TF-IDF later in the post).

But first, let's create the data labels from the Newsgroups categories and a pandas DataFrame of the labels to assist UMAP with plotting.

```
category_labels = [dataset.target_names[x] for x in dataset.target]
hover_df = pd.DataFrame(category_labels, columns=['category'])
```

Create vectors from the text data

The first step to working with text data is to turn it into features. This is a process known as vectorization, as it is mapping text data into a vector form.

There are different ways of vectorization. We are going to start with the simplest, the word-count vectorizer. It uses a "bag-of-words" approach to handle the text data. This is a word-order independent approach that simply counts how many times a particular word appears in a document. We are going to add an additional requirement that a word must be seen at least 5 times to be part of the generated encoded representation of a post. We are going to use Scikit-learn's built-in CountVectorizer to do this.

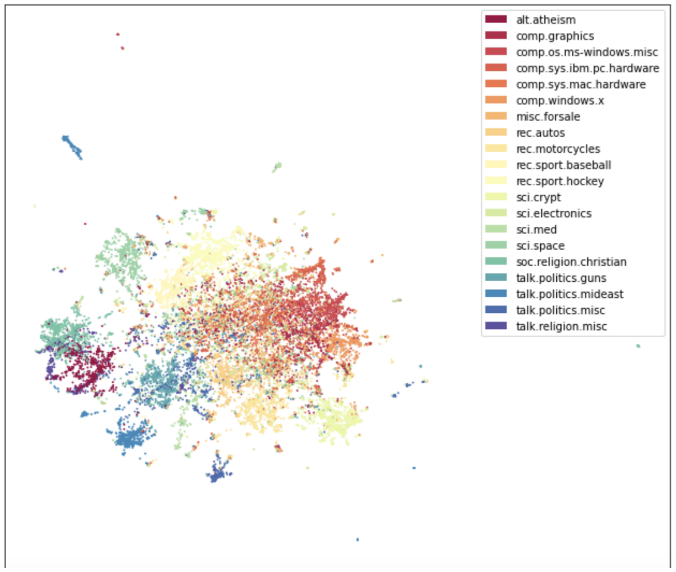
```
vectorizer = CountVectorizer(min_df=5, stop_words='english')
word_doc_matrix = vectorizer.fit_transform(dataset.data)
```

We utilize UMAP configured with the cosine distance measurement. UMAP is a general-purpose dimensionality reduction algorithm. We run it in unsupervised mode, and it functions to create an easily visualized 2D representation of our documents. Cosine distance measurement gives a measure of how similar two documents are to each other. There are different metrics for measuring similarity, and cosine distance is empirically more effective in most NLP tasks than other metrics such as Euclidean distance. We see how our documents, labeled by the different topics, cluster together.

```
embedding = umap.UMAP(n_components=2, metric='cosine').fit(word_doc_matrix)
```

We generate a 2D repersentation of these forum posts for visualization purposes

```
f = umap.plot.points(embedding, labels=hover_df['category'])
```



We can see if we can make improvement to the clustering of the documents. We can do some basic preprocessing steps to improve the embedding of this data. Often, this needs to be done manually, but there are already pre-processing steps built into Scikit-learn for this particular dataset. We begin by removing the unnecessary headers, footers, and quotes.

```
dataset2 = fetch_20newsgroups(subset='all',
                              shuffle=True, random_state=24, remove = ("headers", "footers", "quotes"))
```

Let’s see what one post looks like now.

```
Category: rec.sport.hockey
---------------------------



That is an exceptionally cool idea.
Would it work?

How strong a field is needed?
Anyone?
```

TF-IDF stands for term frequency, inverse document frequency. It gives less weight to words that appear frequently across a large number of documents, since they are more popular in general. It asserts a higher weight to words that appear frequently in a smaller subset of documents, since they are probably important words for those documents. It is another method for vectorizing text data. We utilize TF-IDF to vectorize our posts and specify that stop words (common words such as the, and) should be removed.

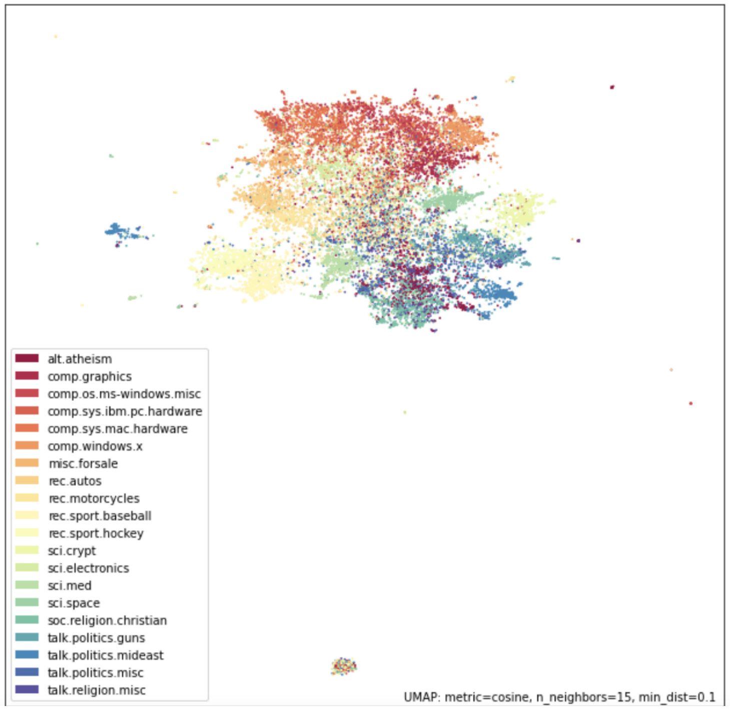
```
tfidf_vectorizer = TfidfVectorizer(min_df=5, stop_words='english')
```

```
tfidf_word_doc_matrix = tfidf_vectorizer.fit_transform(dataset2.data)
```

We create a new set of embeddings and plot it.

```
embedding2 = umap.UMAP(n_components=2, metric='cosine').fit(tfidf_word_doc_matrix)
```

```
f = umap.plot.points(embedding2, labels=hover_df['category'])
```



New embeddings

Quantifying the quality of an embedding is ultimately a subjective task. In general, we can conclude that embeddings with more separation between our labels are more "desirable" in the sense that it is easier to draw a function to separate the labels if the embeddings are more separated. For this reason, it is desirable to see "clumpyness" and the appearance of clusters because that means that the model has meaningfully learned things about the data in an unsupervised fashion.

We see that the addition of TF-IDF along with the preprocessing of the dataset resulted in a significantly more clumpy and reasonable looking embedding. This implies that performance of a classifier which utilized the UMAP representation as its inputs would also likely improve.

Often times, documents do not come with labels. The task is to find ways to group similar documents together. We can use the k-means clustering technique for this.

Group together similar documents using k-means clustering

We will perform clustering using the TF-IDF vectors we have created. This creates basically a clustering that only accounts for the TF-IDF weighted word counts within the text. This does not take text ordering into account yet. It's still pretty good.

Since clustering is basically asking the computer to "come up with the labels", it would not be interesting to find only 10 clusters. Let's ask the computer to split up the dataset into a finer granularity than it already is, by telling it to give us 20 clusters.

```
import sklearn.cluster as cluster
import numpy as np
```

```
kmeans_labels = cluster.KMeans(n_clusters=40).fit_predict(tfidf_word_doc_matrix)
```

We get a list of labels which corresponds to which cluster a document belongs to.

```
kmeans_labels
```

```
array([22, 22, 22, ..., 22, 29, 31], dtype=int32)
```

We use the function below to get examples out of the original dataset that are part of a particular cluster.

```
def get_cluster_examples(data, label_list, label, number_of_examples):
    hits = 0
    for idx, document in enumerate(data):
        if kmeans_labels[idx] == label:
            if hits < number_of_examples:
                category = dataset2.target_names[dataset2.target[idx]]
                print(f'Category: {category}')
                print('---------------------------')
                print(document)
                print('---------------------------')
                hits += 1
```

We can look at a few articles from the fifth cluster.

```
Category: sci.crypt
---------------------------
Fascinating.  Most of the content of the White House announcements was
in what was *not* said.  It gives us almost nothing of value, threatens to
take away a lot, and does it with a sincere smile on its face,
and the nice friendly word "Management".

                  FACT SHEET
             PUBLIC ENCRYPTION MANAGEMENT

The first thing it doesn't say is "We're giving you stronger encryption".
what it says is
   the U. S. Government has developed a microcircuit that not only
   provides privacy through encryption that is substantially more robust
   than the current government standard, but also permits escrowing of
   the keys needed to unlock the encryption.  The system for the
   escrowing of keys will allow the government to gain access to
   encrypted information only with appropriate legal authorization.
```

```
But DES is strong enough that only the government can break it now,
so the major effect is to make it EASIER for government to break!
*At best* it makes it more difficult for the NSA to break, since they
need to get one of the two escrowed keys to do a brute-force search
for the other 40-bit key.
```

```
----------------------------
Category: sci.crypt
----------------------------
I want to add link encryption to a module that multiplexes upper
level routines into a single data link. The upper levels won't know
about this, and thus key exchange shall only need to occur once (at
the initial link establishment). I figure that I can do this with
DES and a Diffie-Hellman key exchange.

Is using a Diffie-Hellman exchange to establish a 64 bit DES key
acceptable, in other words, what are the pro's and con's of such a
setup? Are there any important issues to watch out for (aside from
filtering out unacceptable keys)?

And in order to achieve this, I guess I will need to use 64bit math
routines (for probable prime number calculation, exponentiation etc),
so could someone point me towards a good package (this is strictly
non-commercial).

Matthew.
----------------------------
```

```
----------------------------
Category: sci.crypt
----------------------------
From: Marc VanHeyningen <mvanheyn@cs.indiana.edu>

    The majority of the discussion involving this "Clipper Chip" seems to
    pertain to the encryption of telephone conversations.  Does anyone
    know if that means this chip is designed to work primarily with analog
    signals?  The language sort of suggests this, but it's hard to say.

I'd lay a few bucks that its just data-in-data-out in parallel.  I suspect
to make it a phone you'd need a codec and speech compression.  There would
be a loss of bandwidth on the speech signal, which I suspect would scupper
any of the suggestions I've seen here about putting a different encryption
front end on it.

There's no hint of any modulation scheme in the docs.  I'm sure it's purely
a digital chip.  The back end will come later, but I'm *positive* it won't
be left to the manufacturers - they all have to be the same modulation
scheme to make it easy for the NSA to tap them.

The only other possibility is that this is intended only for ISDN phones.
(Puts a whole new spin on EFFs obsession about ISDN if true, bwahahaha! ;-) )
----------------------------
```

It looks like the fifth cluster contains some of the posts in the cryptography category.

We can also demonstrate the power of Transformer Language models using a simple but strong BERT based paraphrasing model. This model stake word order and semantic meaning into account beyond what TF-IDF is capable of. This model will create a different vector representation of our text data.

```python
from sentence_transformers import SentenceTransformer
transformer_model = SentenceTransformer('paraphrase-distilroberta-base-v1')
```

We need to encode our data again.

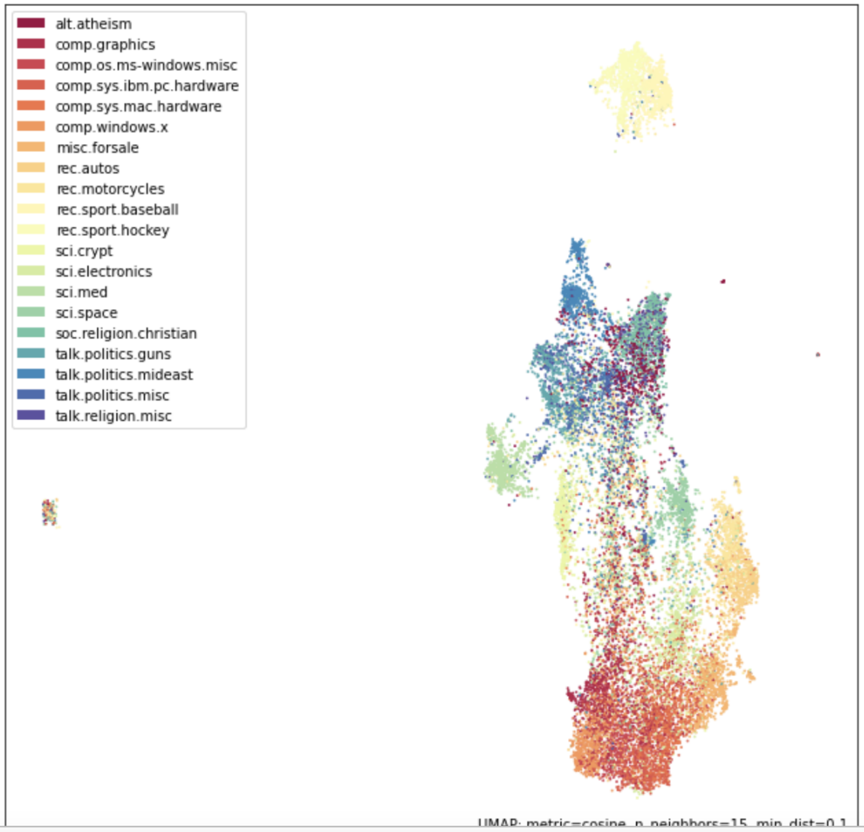
```python
sentence_embeddings = transformer_model.encode(dataset2.data)
```

We can plot the new embeddings and compare to the ones generated from TF-IDF.

```
embedding3 = umap.UMAP(n_components=2, metric='cosine').fit(sentence_embeddings)
```

It is actually quite similar to the one generated from TF-IDF.

```
g = umap.plot.points(embedding3, labels=hover_df['category'])
```



With new embeddings, we can also use them to create new clustering of documents.

Conclusion

In summary, the new conda packs for natural language processing will enable data scientists to use our data science platform to work with text data, including the state-of-the-art Transformers library from Hugging Face.

We showed an example of how to group documents from the 20 Newsgroups dataset. We then demonstrated how to pre-process the data, create vectors from the text data using Count Vectorizer, TF-IDF, and a Transformer-based model for word embeddings, and visualize the grouping of the posts. We also used k-means clustering to group the posts and look at how the method group compared to the labels already assigned to the

the posts. We also used k-means clustering to group the posts and look at how the method group compared to the labels already assigned to the posts.

Explore Oracle's data science platform

Wendy Yip
Data Scientist

Previous Post

Using AI and ML to enrich metadata in Oracle Cloud Infrastructure Data Catalog

Rashmi Badan | 6 min read

Next Post

Accelerate your model build process with the Intel® Extension for Scikit-learn

Praveen Patil | 3 min read