DATA SCIENCE

How to Cluster Documents Using Word2Vec and K-means

Learn how to cluster documents using Word2Vec. In this tutorial, you'll train a Word2Vec model, generate word embeddings, and use K-means to create groups of news articles.

DYLAN CASTILLO
18 JAN 2021 • 15 MIN READ

Table of Contents

I bet you can find more sentiment analysis tutorials online than people doing sentiment analysis in their day jobs. Don't get me wrong. I'm not saying those tutorials aren't useful. I just want to highlight that **supervised learning** receives a lot more attention than any other method in Natural Language Processing (NLP).

Oddly enough, there's a big chance that most of the text data you'll use in your next projects won't have ground truth labels. So supervised learning might not be a solution you can apply to your data problems right away.

What can you do then? Use **unsupervised learning** algorithms.

In this tutorial, you'll learn to apply unsupervised learning to generate value from your text data. You'll cluster documents by training a word embedding (Word2Vec) and applying the K-means algorithm.

Please be aware that the next sections focus on practical manners. You won't find much theory in them, besides brief definitions of relevant ideas.

To make the most of this tutorial, you should be familiar with these topics:

- Supervised and unsupervised learning
- Clustering (particularly, K-means)
- Word2Vec

How to Cluster Documents

You can think of the process of clustering documents in three steps:

1. **Cleaning and tokenizing data:** this usually involves lowercasing text, removing non-alphanumeric characters, or stemming words.
2. **Generating vector representations of the documents**: this concerns the mapping of documents from words into numerical vectors—some common ways of doing this include using bag-of-words models or word embeddings.
3. **Applying a clustering algorithm on the document vectors**: this requires selecting and applying a clustering algorithm to find the best possible groups using the document vectors. Some frequently used algorithms include K-means, DBSCAN, or Hierarchical Clustering.

That's it! Now, you'll see how that looks in practice.

Sample Project: Clustering News Articles

In this section, you'll learn how to cluster documents by working through a small project. You'll group news articles into categories using a dataset published by Szymon Janowski.

Set Up Your Local Environment

To follow along with the tutorial examples, you'll need to download the data and install a few libraries. You can do it by following these

2. Create a new virtual environment using `venv` or `conda`.
3. Activate your new virtual environment.
4. Install the required libraries.
5. Start a Jupyter notebook.

If you're using `venv`, then you need to run these commands:

```
$ git clone https://github.com/dylanjcastillo/nlp-sr
$ python3 -m venv venv
$ source venv/bin/activate
(venv) $ pip install -r requirements
(venv) $ jupyter notebook
```

If you're using `conda`, then you need to run these commands:

```
$ git clone https://github.com/dylanjcastillo/nlp-sr
$ conda create --name venv
$ conda activate venv
(venv) $ pip install -r requirements
(venv) $ jupyter notebook
```

Next, open Jupyter Notebook. Then, create a new notebook in the root folder and set its name to `clustering_word2vec.ipynb`.

By now, your project structure should look like this:

```
nlp-snippets/
|
```

```
├── data/
|
├── ds_utils/
|
├── preprocessing/
|
├── venv/ # (If you're using venv)
|
├── clustering_word2vec.ipynb
├── LICENSE
├── README.md
└── requirements.txt
```

This is your project's structure. It includes these directories and files:

- `clustering/` : Examples of clustering text data using bag-of-words, training a word2vec model, and using a pretrained fastText embeddings.
- `data/` : Data used for the clustering examples.
- `ds_utils/` : Common utility functions used in the sample notebooks in the repository.
- `preprocessing/` : Code snippets frequently used for preprocessing text.
- `venv/` : If you used `venv` , then this directory will contain the files related to your virtual environment.
- `requirements.txt` : Libraries used in the examples provided.
- `README` and `License` : Information about the repository, and its license.

For now, you'll use the notebook you created (`clustering_word2vec.i`

useful for NLP tasks. You can review those on your own.

In the next section, you'll create the whole pipeline from scratch. If you'd like to download the full — and cleaner — version of the code in the examples, then go to the NLP Snippets repository.

That's it for set up! Next, you'll define your imports.

Import the Required Libraries

Once you finish setting up your local environment, it's time to start writing code in your notebook. Open `clustering_word2vec.ipynb`, and copy the following code in the first cell:

```
import os
import random
import re
import string

import nltk
import numpy as np
import pandas as pd

from gensim.models import Word2Vec

from nltk import word_tokenize
from nltk.corpus import stopwords

from sklearn.cluster import MiniBatchKMeans
from sklearn.metrics import silhouette_samples,

nltk.download("stopwords")
```

```
random.seed(SEED)
os.environ["PYTHONHASHSEED"] = str(SEED)
np.random.seed(SEED)
```

These are the libraries you need for the sample project. Here's what you do with each of them:

- **`os`** and **`random`** help you define a random seed to make the code deterministically reproducible.
- **`re`** and **`string`** provide you with easy ways to clean the data.
- **`pandas`** helps you read the data.
- **`numpy`** provides you with linear algebra utilities you'll use to evaluate results. Also, it's used for setting a random seed to make the code deterministically reproducible.
- **`gensim`** makes it easy for you to train a word embedding from scratch using the `Word2Vec` class.
- **`nltk`** aids you cleaning and tokenizing data through the use of the `word_tokenize` method and the `stopword` list.
- **`sklearn`** gives you an easy interface to the clustering model, `MiniBatchKMeans`, and the metrics to evaluate the quality of its results, `silhouette_samples` and `silhouette_score`.

In addition to importing the libraries, you download English stopwords using `nltk.download("stopwords")`, you define `SEED` and set it as random seed using `numpy`, `random`, and the `PYTHONHASHSEED` environment variable. This last step makes sure your code is reproducible across systems.

Run this cell and make sure you don't get any errors. In the next

Clean and Tokenize Data

After you import the required libraries, you need to read and preprocess the data you'll use in your clustering algorithm. The preprocessing consists of cleaning and tokenizing the data. To do that, copy the following function in a new cell in your notebook:

```
def clean_text(text, tokenizer, stopwords):
    """Pre-process text and generate tokens

    Args:
        text: Text to tokenize.

    Returns:
        Tokenized text.
    """
    text = str(text).lower()  # Lowercase words
    text = re.sub(r"\[(.*?)\]", "", text)  # Re
    text = re.sub(r"\s+", " ", text)  # Remove
    text = re.sub(r"\w+…|…", "", text)  # Remov
    text = re.sub(r"(?<=\w)-(?=\w)", " ", text)
    text = re.sub(
        f"[{re.escape(string.punctuation)}]", "
    )  # Remove punctuation

    tokens = tokenizer(text)  # Get tokens from
    tokens = [t for t in tokens if not t in sto
    tokens = ["" if t.isdigit() else t for t in
    tokens = [t for t in tokens if len(t) > 1]
    return tokens
```

This code cleans and tokenizes a `text` input, using a predefined `tokenizer` and a list of `stopwords`. It helps you perform these operations:

2. **Line 11:** Remove substrings like "[+300 chars]" that I found while reviewing the data.
3. **Line 12:** Remove multiple spaces, tabs, and line breaks.
4. **Line 13:** Remove ellipsis characters.
5. **Lines 14-17:** Replace dashes between words with a space and remove punctuation.
6. **Lines 19-20:** Tokenize text and remove tokens using a list of stop words.
7. **Lines 21-22:** Remove digits and tokens which length is too short.

Then, in the next cell, copy the following code to read the data and apply that function to the text columns:

```
custom_stopwords = set(stopwords.words("english
text_columns = ["title", "description", "conten

df = df_raw.copy()
df["content"] = df["content"].fillna("")

for col in text_columns:
    df[col] = df[col].astype(str)

# Create text column based on title, descriptio
df["text"] = df[text_columns].apply(lambda x: "
df["tokens"] = df["text"].map(lambda x: clean_t

# Remove duplicated after preprocessing
_, idx = np.unique(df["tokens"], return_index=T
```

```
# Remove empty values and keep relevant columns
df = df.loc[df.tokens.map(lambda x: len(x) > 0)]

print(f"Original dataframe: {df_raw.shape}")
print(f"Pre-processed dataframe: {df.shape}")
```

This is how you read and preprocess the data. This code applies the cleaning function you defined earlier, removes duplicates and nulls, and drops irrelevant columns.

You apply these steps to a new dataframe (`df`) you'll use for the next examples. It contains a column with the raw documents called `text` and another one with the preprocessed documents called `tokens` .

If you execute the two cells you defined, then you should get the following output:

```
Original dataframe: (10437, 15)
Pre-processed dataframe: (9882, 2)
```

Next, you'll create document vectors using Word2Vec.

Generate Document Vectors

After you've cleaned and tokenized the text, you'll use the documents' tokens to create vectors using Word2Vec. This process consists of two steps:

1. Train a Word2Vec model using the tokens you generated earlier. Alternatively, you could load a pre-trained Word2Vec model (I'll also show you how to do it).

In this section, you'll go through these steps.

Train Word2Vec Model

The following code will help you train a Word2Vec model. Copy it into a new cell in your notebook:

```
model = Word2Vec(sentences=tokenized_docs, vect
```

You use this code to train a Word2Vec model based on your tokenized documents. For this example, you specified the following parameters in the `Word2Vec` class:

- `sentences` expects a list of lists with the tokenized documents.
- `vector_size` defines the size of the word vectors. In this case, you set it to 100.
- `workers` defines how many cores you use for training. I set it to 1 to make sure the code is deterministically reproducible.
- `seed` seed for random number generator. It's set to the constant `SEED` you defined in the first cell.

There's other parameters you can tune when training the Word2Vec model. Take a look at gensim's documentation if you'd like to learn more about them.

> **Note:** In many cases, you might want to use a pre-trained model instead of training one yourself. If that's the case, gensim provides you with an easy way to access some of the most

You can load a pre-trained Word2Vec model as follows:

```
wv = api.load('word2vec-google-news-300')
```

One last thing, if you're following this tutorial and decide to use a pre-trained model, you'll need to replace `model.wv` by `wv` in the code snippets from here on. Otherwise, you'll get an error.

Next, run the cell you just created in your notebook. It might take a couple of minutes. After its done, you can validate that the results make sense by plotting the vectors or by reviewing the similarity results for relevant words. You can do the latter by copying and running this code in a cell in your notebook:

```
model.wv.most_similar("trump")
```

If you run this code, then you'll get this output:

```
[('trumps', 0.988541841506958),
 ('president', 0.9746493697166443),
 ('donald', 0.9274922013282776),
 ('ivanka', 0.9203903079032898),
 ('impeachment', 0.9195784330368042),
 ('pences', 0.9152231812477112),
 ('avlon', 0.9148306846618652),
 ('biden', 0.9146010279655457),
 ('breitbart', 0.9144087433815002),
 ('vice', 0.9067237973213196)]
```

That's it! You've trained your Word2Vec model, now you'll use it to

Create Document Vectors from Word Embedding

Now you'll generate document vectors using the Word2Vec model you trained. The idea is straightforward. From the Word2Vec model you'll get numerical vectors per each of the words in a document, so you need to find a way of generating a single vector out of them.

A common approach is to use the average of the vectors. This approach works well for short texts. For longer texts, there's not a clear consensus what will work well. Though, using a weighted average the vectors might help.

The following code will help you create a vector per document by averaging its word vectors. Create a new cell in your notebook and copy this code there:

```
def vectorize(list_of_docs, model):
    """Generate vectors for list of documents u

    Args:
        list_of_docs: List of documents
        model: Gensim's Word Embedding

    Returns:
        List of document vectors
    """
    features = []

    for tokens in list_of_docs:
        zero_vector = np.zeros(model.vector_siz
        vectors = []
```

```
                    try:
                        vectors.append(model.wv[tok
                    except KeyError:
                        continue
            if vectors:
                vectors = np.asarray(vectors)
                avg_vec = vectors.mean(axis=0)
                features.append(avg_vec)
            else:
                features.append(zero_vector)
        return features

    vectorized_docs = vectorize(tokenized_docs, mod
    len(vectorized_docs), len(vectorized_docs[0])
```

This code will get all the word vectors of each document and average them to generate a vector per each document. Here's what's happening there:

1. You define the `vectorize` function that takes a list of documents and a `gensim` model as input, and generates a feature vector per document as output.
2. You apply the function to the documents' tokens in `tokenized_doc`, using the Word2Vec `model` you trained earlier.
3. You print the length of the list of documents and the size of the generated vectors.

Next, you'll cluster the documents using Mini-batches K-means.

Cluster Documents Using (Mini-batches) K-means

To cluster the documents, you'll use the **Mini-batches K-means**

shares the same objective function with the original algorithm so, in practice, the results are just a bit worse than K-means.

In the code snippet below, you can see the function you'll use to create the clusters using Mini-batches K-means. Create a new cell in your notebook, and copy the following code there:

```
def mbkmeans_clusters(
    X,
    k,
    mb,
    print_silhouette_values,
):
    """Generate clusters and print Silhouette m

    Args:
        X: Matrix of features.
        k: Number of clusters.
        mb: Size of mini-batches.
        print_silhouette_values: Print silhouet

    Returns:
        Trained clustering model and labels bas
    """
    km = MiniBatchKMeans(n_clusters=k, batch_si
    print(f"For n_clusters = {k}")
    print(f"Silhouette coefficient: {silhouette_
    print(f"Inertia:{km.inertia_}")

    if print_silhouette_values:
        sample_silhouette_values = silhouette_s
        print(f"Silhouette values:")
```

```
            cluster_silhouette_values = sample_
            silhouette_values.append(
                (
                    i,
                    cluster_silhouette_values.s
                    cluster_silhouette_values.m
                    cluster_silhouette_values.m
                    cluster_silhouette_values.m
                )
            )
        silhouette_values = sorted(
            silhouette_values, key=lambda tup:
        )
        for s in silhouette_values:
            print(
                f"    Cluster {s[0]}: Size:{s[1
            )
    return km, km.labels_
```

This function creates the clusters using the Mini-batches K-means algorithm. It takes the following arguments:

- **`X`** : Matrix of features. In this case, it's your vectorized documents.
- **`k`** : Number of clusters you'd like to create.
- **`mb`** : Size of mini-batches.
- **`print_silhouette_values`** : Defines if the Silhouette Coefficient is printed for each cluster. If you haven't heard about this coefficient, don't worry, you'll learn about it in a bit!

`mbkmeans_cluster` takes these arguments and returns the fitted

function to your vectorized documents.

Definition of Clusters

Now, you need to execute `mbkmean_clusters` providing it with the vectorized documents, and the number of clusters. You'll print the Silhouette Coefficients per cluster, to review the quality of your clusters.

Create a new cell and copy this code there:

```
clustering, cluster_labels = mbkmeans_clusters(
    X=vectorized_docs,
    k=50,
    mb=500,
    print_silhouette_values=True,
)
df_clusters = pd.DataFrame({
    "text": docs,
    "tokens": [" ".join(text) for text in token
    "cluster": cluster_labels
})
```

This code will fit the clustering model, print the Silhouette Coefficient per cluster, and return the fitted model and the labels per cluster. It'll also create a dataframe you can use to review the results.

There's a few things to consider when setting the input arguments:

- `print_silhouette_values` is straightforward. In this case, you set it to `True` to print the evaluation metric per cluster. This

sure that it is not too small to avoid a significant impact on the quality of results and not too big to avoid making the execution too slow. In this case, you set it to 500 observations.

- `k` is trickier. In general, it involves a mix of qualitative analysis and quantitative metrics. After a few experiments on my side, I found that 50 seemed to work well. But that is more or less arbitrary.

For the quantitative evaluation of the number of clusters, you could use metrics like the **Silhouette Coefficient**. This coefficient is an evaluation metric frequently used in problems where ground truth labels are not known. It's calculated using the mean intra-cluster distance and the mean nearest-cluster distance, and goes from -1 to 1. Well defined clusters result in positive values of this coefficient, while incorrect clusters will result in negative values. If you'd like to learn more about it, then take a look at scikit-learn's documentation.

The qualitative part generally requires you to have domain knowledge of the subject matter so that you can sense-check the results of your clustering algorithm. In the next section, I'll show you two approaches you can use to do a qualitative check of your results.

After executing the cell you just created, the output should look like this:

```
For n_clusters = 50
Silhouette coefficient: 0.11
Inertia:3568.342791047967
Silhouette values:
    Cluster 29: Size:50 | Avg:0.39 | Min:0.01 | Max:
```

```
    Cluster 39: Size:81 | Avg:0.31 | Min:-0.05 | Max
    Cluster 27: Size:63 | Avg:0.28 | Min:0.02 | Max:
    Cluster 6: Size:101 | Avg:0.27 | Min:0.02 | Max:
    Cluster 24: Size:120 | Avg:0.26 | Min:-0.04 | Ma
    Cluster 49: Size:65 | Avg:0.26 | Min:-0.03 | Max
    Cluster 47: Size:53 | Avg:0.23 | Min:0.01 | Max:
    Cluster 22: Size:78 | Avg:0.22 | Min:-0.01 | Max
    Cluster 45: Size:38 | Avg:0.21 | Min:-0.07 | Max
...
```

This is the output of your clustering algorithm. The sizes and Silhouette Coefficients per cluster are the most relevant metrics. The clusters are printed by the value of the Silhouette coefficient in descending order. A higher score means denser – and thus better – clusters. In this case, you can see that clusters 29, 35, and 37 seem to be the top ones.

Next, you'll learn how to check what's in each cluster.

Qualitative Review of Clusters

There are a few ways you can do a qualitative analysis of the results. During the earlier sections, the approach you took resulted in vector representations of tokens and documents and vectors of the clusters' centroids. To analyze the results, you can find the most representative tokens and documents by looking for the vectors closest to the clusters' centroids.

Here's how you obtain the most representative tokens per cluster:

```
1 print("Most representative terms per cluster (b
2 for i in range(50):
```

```
        for t in most_representative:
            tokens_per_cluster += f"{t[0]} "
        print(f"Cluster {i}: {tokens_per_cluster}")
```

For the top clusters we identified earlier – 29, 35, and 37 – these are the results:

```
Cluster 29: noaa sharpie claim assertions forecaster
Cluster 35: eye lilinow path halts projected
Cluster 37: cnnpolitics complaint clinton pences whi
```

Next, we can do the same analysis but with documents instead of tokens. This is how you find the most representative documents for cluster 29:

```
test_cluster = 29
most_representative_docs = np.argsort(
    np.linalg.norm(vectorized_docs - clustering
)
for d in most_representative_docs[:3]:
    print(docs[d])
    print("-------------")
```

And these are the 3 most representative documents in that cluster:

```
Dorian, Comey and Debra Messing: What Trump tweeted
-------------
Ross Must Resign If Report He Threatened NOAA Officia
"If that story is true, and I don't… [+3828 chars]
-------------
```

Most of the results seem to be related to a dispute between Donald Trump and the National Oceanic and Atmospheric Agency (NOAA). It was a famous controversy that people referred to as Sharpiegate.

You could also explore other approaches like generating word frequencies per cluster or review random samples of documents per cluster.

Other Approaches

There are other approaches you could take to cluster text data like:

- Using a pre-trained word embedding instead of training your own. In this tutorial, you trained a Word2Vec model from scratch but it's very common to use a pre-trained model.
- Generating feature vectors using a bag-of-words approach instead of word embeddings.
- Reducing dimensionality of feature vectors. This is very useful if you use a bag-of-words approach.
- Clustering documents using other algorithms like HDBSCAN, or Hierarchical Clustering.
- Using BERT sentence embeddings to generate the feature vectors. Or generating the topics with BERTopic.

Conclusion

Way to go! You just learned how to cluster documents using Word2Vec. You went through an end-to-end project, where you learned all the steps required for clustering a corpus of text.

- **Preprocess** data to use with a Word2Vec model
- **Train** a Word2Vec model
- Use quantitative metrics, like the **Silhouette score**, to evaluate the quality of your clusters
- Find the most **representative tokens** and **documents** in your clusters

I hope you find this tutorial useful. Shoot me a message if you have any questions!

4 Comments - *powered by utteranc.es*

paddyxxx commented 2 months ago

Hello Dylan! Thank you for the tutorial - it is extremely useful. However, I have one tiny problem. I am trying to run the top 100 predictive words through a pretrained word2vec model with this code and unfortunately I am not getting the right results. I am creating a variable list_of_docs which consists of all the 100 words and tokenized_docs which consists of the tokenized data from the 100 predictive words - and then I am running this code. However, it seems that the 100 words really do not cross through the word2vec as the cluster output I get consists of words which are probably contained in the pretrained Wikipedia word2vec model but not of my 100 words. Do you have many any idea where I am going wrong? - Thank you for your help in advance! I appreciate it.

paddyxxx commented 2 months ago

- just a clarifying comment - the top 100 predictive words from my previously done bag-of-words analysis

dylanjcastillo commented 2 months ago Owner

Hi @paddyxxx,

Thank you. I'm glad you found it useful!

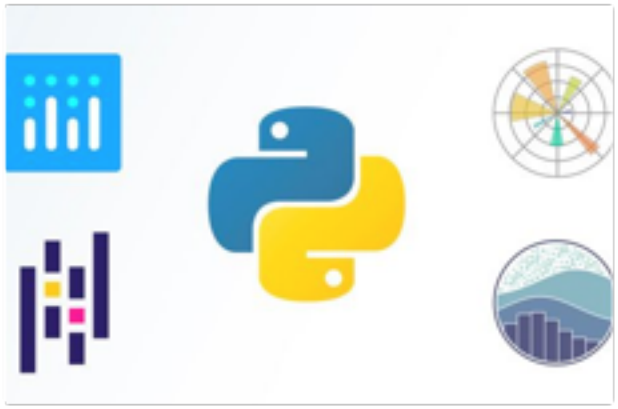
MORE IN DATA SCIENCE

Clean and Tokenize Text With Python
10 Dec 2020 – 9 min read

4 Ways To Improve Your Graphs Using Plotly
25 May 2020 – 8 min read

Fast & Asynchronous In Python: Accelerate Your Requests Using asyncio
1 Mar 2020 – 8 min read

See all 4 posts →



How to Plot with Python: 8 Popular Graphs Made with pandas, matplotlib, seaborn, and plotly.express

In this tutorial, you'll learn how to make some of the most popular types of charts with four data visualization libraries: pandas, matplotlib, seaborn, and plotly.express.

DYLAN CASTILLO
3 JUL 2021 • 27 MIN READ

DATA SCIENCE

Clean and Tokenize Text With Python

The first step in a Machine Learning project is cleaning the data. In this article, you'll find 20 code snippets to clean and tokenize text data using Python.

DYLAN CASTILLO
10 DEC 2020 • 9 MIN READ

