10+ Examples for Using CountVectorizer

 kavita-ganesan.com/how-to-use-countvectorizer

December 5, 2019

Scikit-learn's CountVectorizer is used to transform a corpora of text to a vector of term / token counts. It also provides the capability to preprocess your text data prior to generating the vector representation making it a highly flexible feature representation module for text.

In this article, we are going to go in-depth into the different ways you can use CountVectorizer such that you are not just computing counts of words, but also **preprocessing** your text data appropriately as well as **extracting additional features** from your text dataset.

Example of How CountVectorizer Works

To show you an example of how CountVectorizer works, let's take the book title below (for context: this is part of a book series that kids love) :

```
doc=["One Cent, Two Cents, Old Cent, New Cent: All About Money"]
```

This text is transformed to a sparse matrixas shown in Figure 1(b) below:

	about	all	cent	cents	money	new	old	one	two
doc	1	1	3	1	1	1	1	1	1

In theory (a)

Index	0	1	2	3	4	5	6	7	8
doc	1	1	3	1	1	1	1	1	1

In practice (b)

Figure 1: CountVectorizer sparse matrix representation of words. (a) is how you visually think about it. (b) is how it is really represented in practice.

Notice that here we have **9 unique words**. So 9 columns. Each column in the matrix represents a unique word in the vocabulary, while each row represents the document in our dataset. In this case, we only have one book title (i.e. the document), and therefore we have only 1 row. The values in each cell are the word counts. Note that with this representation, counts of some words could be 0 if the word did not appear in the corresponding document.

While visually it's easy to think of a word matrix representation as Figure 1 (a), in reality, these words are transformed to numbers and these numbers represent positional index in the **sparse matrix as seen in Figure 1(b)**.

Why the sparse matrix format?

With CountVectorizer we are converting raw text to a numerical vector representation of words and n-grams. This makes it easy to directly use this representation as features (signals) in Machine Learning tasks such as for text classification and clustering.

Note that these algorithms only understand the concept of *numerical features* irrespective of its underlying type (text, image pixels, numbers, categories and etc.) allowing us to perform complex machine learning tasks on different types of data.

Side Note: If all you are interested in are word counts, then you can get away with using the python Counter. There is no real need to use CountVectorizer. However, if you still want to use CountVectorizer, here's the example for extracting counts with CountVectorizer.

Dataset & Imports

In this tutorial, we will be using titles of 5 cat in the hat books (as seen below).

```
from sklearn.feature_extraction.text import CountVectorizer
cat_in_the_hat_docs=[
       "One Cent, Two Cents, Old Cent, New Cent: All About Money (Cat in the Hat's
Learning Library",
       "Inside Your Outside: All About the Human Body (Cat in the Hat's Learning
Library)",
       "Oh, The Things You Can Do That Are Good for You: All About Staying Healthy
(Cat in the Hat's Learning Library)",
       "On Beyond Bugs: All About Insects (Cat in the Hat's Learning Library)",
       "There's No Place Like Space: All About Our Solar System (Cat in the Hat's
Learning Library)"
      ]
```

I had intentionally made it a handful of short texts so that you can see how to put **CountVectorizer** to full use in your applications. Keep note that each title above is considered a document.

CountVectorizer Plain and Simple

```
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer(cat_in_the_hat_docs)
count_vector=cv.fit_transform(cat_in_the_hat_docs)
```

What happens above is that the 5 books titles are preprocessed, tokenized and represented as a sparse matrix as explained in the introduction. By default, CountVectorizer does the following:

- lowercases your text (set `lowercase=false` if you don't want lowercasing)
- uses utf-8 encoding
- performs tokenization (converts raw text to smaller units of text)
- uses word level tokenization (meaning each word is treated as a separate token)
- ignores single characters during tokenization (say goodbye to words like 'a' and 'I')

Now, let's look at the vocabulary (collection of unique words from our documents):

```
# show resulting vocabulary; the numbers are not counts, they are the position in the
sparse vector.
cv.vocabulary_

#shape of count vector: 5 docs (book titles) and 43 unique words
count_vector.shape

(5,43)
```

We have 5 (rows) documents and 43 unique words (columns)!

CountVectorizer and Stop Words

Now, the first thing you may want to do, is to eliminate stop words from your text as it has limited predictive power and may not help with downstream tasks such as text classification. Stop word removal is a breeze with CountVectorizer and it can be done in several ways:

1. Use a custom stop word list that you provide
2. Use sklearn's built in English stop word list (not recommended)
3. Create corpora specific stop words using `max_df` and `min_df` (highly recommended and will be covered later in this tutorial)

Let's look at the 3 ways of using stop words.

Custom Stop Word List

```
cv = CountVectorizer(cat_in_the_hat_docs,stop_words=["all","in","the","is","and"])
count_vector=cv.fit_transform(cat_in_the_hat_docs)
count_vector.shape

(5,40)
```

In this example, we provide a list of words that act as our stop words. Notice that the shape has gone from `(5,43)` to `(5,40)` because of the stop words that were removed. Note that we can actually load stop words directly from a file into a list and supply that as the stop word

list.

To check the stop words that are being used (when explicitly specified), simply access `cv.stop_words`.

```
# any stop words that we explicitly specified?
cv.stop_words
```

```
['all', 'in', 'the', 'is', 'and']
```

While `cv.stop_words` gives you the stop words that you explicitly specified as shown above, `cv.stop_words_` (**note**: with underscore suffix) gives you the stop words that CountVectorizer inferred from your `min_df` and `max_df` settings as well as those that were cut off during feature selection (through the use of `max_features`). So far, we have not used the three settings, so `cv.stop_words_` will be empty.

Stop Words using MIN_DF

The goal of `MIN_DF` is to ignore words that have very few occurrences to be considered meaningful. For example, in your text you may have names of people that may appear in only 1 or two documents. In some applications, this may qualify as noise and could be eliminated from further analysis.

Instead of using a minimum term frequency (total occurrences of a word) to eliminate words, `MIN_DF` looks at ***how many documents contained a term***, better known as **document frequency**. The `MIN_DF` value can be an **absolute value** (e.g. 1, 2, 3, 4) or a value representing **proportion of documents** (e.g. 0.25 meaning, ignore words that have appeared in 25% of the documents) .

Eliminating words that appeared in less than 2 documents:

```
# ignore terms that appeared in less than 2 documents
cv = CountVectorizer(cat_in_the_hat_docs,min_df=2)
count_vector=cv.fit_transform(cat_in_the_hat_docs)
```

Now, to see which words have been eliminated, you can use `cv.stop_words_` as this was internally inferred by CountVectorizer (see output below).

```
{'are',
 'beyond',
 'body',
 'bugs',
 'can',
 'cent',
 'cents',
 'do',
 'for',
 'good',
 'healthy',
 'human',
 'insects',
 'inside',
 'like',
 'money',
 'new',
 'no',
 'oh',
 'old',
 'on',
 'one',
 'our',
 'outside',
 'place',
 'solar',
 'space',
 'staying',
 'system',
 'that',
 'there',
 'things',
 'two',
 'you',
 'your'}
```

Yikes! We removed everything? Not quite. However, most of our words have become stop words and that's because we have only 5 book titles.

To see what's remaining, all we need to do is check the vocabulary again with `cv.vocabulary_` (see output below):

```
{'all': 1,
 'about': 0,
 'cat': 2,
 'in': 4,
 'the': 7,
 'hat': 3,
 'learning': 5,
 'library': 6}
```

Sweet! These are words that appeared in all 5 book titles.

Stop Words using MAX_DF

Just as we ignored words that were too rare with `MIN_DF`, we can ignore words that are too common with `MAX_DF`. `MAX_DF` looks at ***how many documents contained a term***, and if it exceeds the `MAX_DF` threshold, then it is eliminated from consideration. The `MAX_DF` value can be an **absolute value** (e.g. 1, 2, 3, 4) or a value representing **proportion of documents** (e.g. 0.85 meaning, ignore words appeared in 85% of the documents as they are too common).

```
# ignore terms that appear in 50% of the documents
cv = CountVectorizer(cat_in_the_hat_docs,max_df=0.50)
count_vector=cv.fit_transform(cat_in_the_hat_docs)
```

I've typically used a value from `0.75-0.85` depending on the task and for more aggressive stop word removal you can even use a smaller value.

Now, to see which words have been eliminated, you can use `cv.stop_words_` (see output below):

```
{'about', 'all', 'cat', 'hat', 'in', 'learning', 'library', 'the'}
```

In this example, all words that appeared in all 5 book titles have been eliminated.

Why document frequency for eliminating words?

Document frequency is sometimes a better way for inferring stop words compared to term frequency as term frequency can be misleading. For example, let's say 1 document out of 250,000 documents in your dataset, contains 500 occurrences of the word `catnthehat`. If you use term frequency for eliminating rare words, the counts are so high that it may never pass your threshold for elimination. The word is still rare as it appears in *only* one document.

On several occasions, such as in building topic recommendation systems, I've found that using document frequency for eliminating rare and common terms gives far better results than relying on just overall term frequency.

Custom Tokenization

The default tokenization in CountVectorizer removes all special characters, punctuation and single characters. If this is not the behavior you desire, and you want to keep punctuation and special characters, you can provide a custom tokenizer to CountVectorizer.

In the example below, we provide a custom tokenizer using `tokenizer=my_tokenizer` where `my_tokenizer` is a function that attempts to keep all punctuation, and special characters and tokenizes only based on whitespace.

```
import re
def my_tokenizer(text):
    # create a space between special characters
    text=re.sub("(\\W)"," \\1 ",text)
    # split based on whitespace
    return re.split("\\s+",text)

cv = CountVectorizer(cat_in_the_hat_docs,tokenizer=my_tokenizer)
count_vector=cv.fit_transform(cat_in_the_hat_docs)
print(cv.vocabulary_)

{'one': 34, 'cent': 14, ',': 4, 'two': 47, 'cents': 15, 'old': 32, 'new': 29, ':': 5,
'all': 7, 'about': 6, 'money': 28, '(': 2, 'cat': 13, 'in': 22, 'the': 44, 'hat': 19,
"'": 1, 's': 38, 'learning': 25, 'library': 26, 'inside': 24, 'your': 49, 'outside':
36, 'human': 21, 'body': 10, ')': 3, ....'beyond': 9, 'bugs': 11, 'insects': 23,
'there': 45, 'no': 30, 'place': 37, 'like': 27, 'space': 40, 'our': 35, 'solar': 39,
'system': 42}
```

Fantastic, now we have our punctuation, single characters and special characters!

Custom Preprocessing

In many cases, we want to preprocess our text prior to creating a sparse matrix of terms. As I've explained in my text preprocessing article, preprocessing helps reduce noise and improves sparsity issues resulting in a more accurate analysis.

Here is an example of how you can achieve custom preprocessing with CountVectorizer by setting `preprocessor=<some_preprocessor>` .

```
import re
import nltk
import pandas as pd
from nltk.stem import PorterStemmer
# init stemmer
porter_stemmer=PorterStemmer()
def my_cool_preprocessor(text):

    text=text.lower()
    text=re.sub("\\W"," ",text) # remove special chars
    text=re.sub("\\s+(in|the|all|for|and|on)\\s+"," _connector_ ",text) # normalize
certain words

    # stem words
    words=re.split("\\s+",text)
    stemmed_words=[porter_stemmer.stem(word=word) for word in words]
    return ' '.join(stemmed_words)
cv = CountVectorizer(cat_in_the_hat_docs,preprocessor=my_cool_preprocessor)
count_vector=cv.fit_transform(cat_in_the_hat_docs)
```

In the example above, `my_cool_preprocessor` is a predefined function where we perform the following steps:

1. lowercase the text (note: this is done by default if a custom preprocessor is not specified)
2. remove special characters
3. normalize certain words
4. use stems of words instead of the original form (see: preprocessing article on stemming)

You can introduce your very own preprocessing steps such as lemmatization, adding parts-of-speech and so on to make this preprocessing step even more powerful.

Working With N-Grams

One way to enrich the representation of your features for tasks like text classification, is to use n-grams where n > 1. The intuition here is that bi-grams and tri-grams can capture contextual information compared to just unigrams. In addition, for tasks like keyword extraction, unigrams alone while useful, provides limited information. For example, `good food` carries more meaning than just `good` and `food` when observed independently.

Working with n-grams is a breeze with CountVectorizer. You can use word level n-grams or even character level n-grams (very useful in some text classification tasks). Here are a few examples:

Word level – bigrams only

```
# only bigrams, word level
cv = CountVectorizer(cat_in_the_hat_docs,ngram_range=
(2,2),preprocessor=my_cool_preprocessor)
count_vector=cv.fit_transform(cat_in_the_hat_docs)

{'one cent': 35,
 'cent two': 19,
 'two cent': 47,
 'cent old': 18,
 'old cent': 33,
 'cent new': 17,
 'new cent': 30,
 'cent _connector_': 16,
 '_connector_ about': 0,
 'about money': 7,
 'money cat': 29,...}
```

Word level – unigrams and bigrams

```
# unigrams and bigrams, word level
cv = CountVectorizer(cat_in_the_hat_docs,ngram_range=
(1,2),preprocessor=my_cool_preprocessor)
count_vector=cv.fit_transform(cat_in_the_hat_docs)

{'one': 60,
 'cent': 24,
 'two': 84,
 'old': 56,
 ...
 'one cent': 61,
 'cent two': 28,
 'two cent': 85,
 'cent old': 27,
 'old cent': 57,
 'cent new': 26,
 'new cent': 51,
 ...}
```

Character level – bigrams only

```
#only character level bigrams
cv = CountVectorizer(cat_in_the_hat_docs,ngram_range=(2,2),analyzer='char_wb')
count_vector=cv.fit_transform(cat_in_the_hat_docs)

{' o': 11,
 'on': 77,
 'ne': 67,
 'e ': 36,
 ' c': 3,
 'ce': 30,
 'en': 40,
 'nt': 72,
 't ': 96,
 ' t': 14,
 'tw': 102,
 'wo': 109,
 'o ': 73,
 'ol': 76,
 'ld': 58,
 'd ': 33,
 ' n': 10,
 'ew': 42,
 ...}
```

Limiting Vocabulary Size

When your feature space gets too large, you can limit its size by putting a restriction on the vocabulary size. Say you want a max of `10,000` n-grams. CountVectorizer will keep the top `10,000` most frequent n-grams and drop the rest.

Since we have a toy dataset, in the example below, we will limit the number of features to `10`.

```
#only bigrams and unigrams, limit to vocab size of 10
cv = CountVectorizer(cat_in_the_hat_docs,max_features=10)
count_vector=cv.fit_transform(cat_in_the_hat_docs)

count_vector.shape

(5, 10)
```

Notice that the shape now is `(5,10)` as we asked for a limit of 10 on the vocabulary size. You can check the removed words using `cv.stop_words_`.

Ignore Counts and Use Binary Values

By default, CountVectorizer uses the counts of terms/tokens. However, you can choose to just use presence or absence of a term instead of the raw counts. This is useful in some tasks such as certain features in text classification where the frequency of occurrence is insignificant. To get binary values instead of counts all you need to do is set `binary=True`.

If you set `binary=True` then CountVectorizer no longer uses the counts of terms/tokens. If a token is present in a document, it is 1, if absent it is `0` regardless of its frequency of occurrence. By default, `binary=False`.

```
# unigrams and bigrams, word level
cv = CountVectorizer(cat_in_the_hat_docs,binary=True)
count_vector=cv.fit_transform(cat_in_the_hat_docs)
```

Using CountVectorizer to Extract N-Gram / Term Counts

Finally, you may want to use CountVectorizer to obtain counts of your n-grams. This is slightly tricky to do with CountVectorizer, but achievable as shown below:

```
def sort_coo(coo_matrix):
    tuples = zip(coo_matrix.col, coo_matrix.data)
    return sorted(tuples, key=lambda x: (x[1], x[0]), reverse=True)

def extract_topn_from_vector(feature_names, sorted_items, topn=10):
    """return n-gram counts in descending order of counts"""

    #use only topn items from vector
    sorted_items = sorted_items[:topn]
    results=[]

    # word index, count i
    for idx, count in sorted_items:

        # get the ngram name
        n_gram=feature_names[idx]

        # collect as a list of tuples
        results.append((n_gram,count))
    return results

cv = CountVectorizer(cat_in_the_hat_docs,ngram_range=
(1,2),preprocessor=my_cool_preprocessor,max_features=100)
count_vector=cv.fit_transform(cat_in_the_hat_docs)
#sort the counts of first book title by descending order of counts
sorted_items=sort_coo(count_vector[0].tocoo())
#Get feature names (words/n-grams). It is sorted by position in sparse matrix
feature_names=cv.get_feature_names()
n_grams=extract_topn_from_vector(feature_names,sorted_items,10)
```

The counts are first ordered in descending order. Then from this list, each feature name is extracted and returned with corresponding counts.

Summary

CountVectorizer provides a powerful way to extract and represent features from your text data. It allows you to control your **n-gram size**, perform **custom preprocessing**, **custom tokenization**, eliminate **stop words** and **limit vocabulary size**.

While counts of words can be useful signals by themselves, in some cases, you will have to use alternative schemes such as TF-IDF to represent your features. For some applications, a binary bag of words representation may also be more effective than counts.

For a more sophisticated feature representation, people use word, sentence and paragraph embeddings trained using algorithms like word2vec, Bert and ELMo where each textual unit is encoded using a fixed length vector.

If there is anything that I missed out here, do feel free to leave a comment below. Here are the code samples for you to try out.

See Also: HashingVectorizer vs. CountVectorizer

Resources

Recommended reading

Join 750+ leaders and practitioners.
Get AI tips sent directly to your inbox.