Bag of Words

We need to convert the string representation of the corpus into a numeric representation that we can apply our machine learning algorithms to. We discard most of the structure of the input text, and only count how oftern each word appears in each text in the corpus. This is the mental image of a "bag of words" for a corpus of docs. It consists of three steps:

- Tokenization, (CountVectorizer)
- Vocabulary Building, (fitting the CountVectorizer builds the vocabulary)(vocabulary_)
- Encoding (in the form of SciPy sparse matrix)(boW is created via transform) (one vector of word counts for each document in the corpus)(for each word in the document, we have a count of how often it appears in each document)

CountVectorizer eliminates single letter words like "a"

CountVectorizer

To create a Count Vectorizer, we simply need to instantiate one. We are not using any parameters yet.

Fitting of the CountVectorizer consists of tokenization of the training data, and building the vocabulary. Transforming the CountVectorizer creates the bag-of-words representation of the train data. The bow is stored in a SciPy sparse matrix that only stores the nonzero entries. To look at the actual content of the sparse matrix, convert it to dense array using numpy.toarray() method

In [5]:
```
from sklearn.feature_extraction.text import CountVectorizer

#Let's get a sample text
sample_text = ["I was wondering if anyone out there could enlighten me on thi
               "the other day. It was a 2-door sports car, looked to be from 
               "early 70s. It was called a Bricklin. The doors were really sm
              ]
sample_text_1 = ["One of the most basic ways we can numerically represent wor
               "is through the one-hot encoding method (also sometimes called
               "count vectorizing)."]
sample_text_2 = ['One, one, one, we can Dance! Such a dance!']
```

In [6]:
```
# instantiate the CountVectorizer
vectorizer = CountVectorizer()
```

In [8]:
```
# fit call tokenizes the text, and builds the vocabulary.
vectorizer.fit(sample_text_2)
```

Out[8]:
```
CountVectorizer()
```

In [ ]:
```
# see the vocabulary of your text
```

In [13]:
```
# To actually create the vectorizer, we simply need to call fit on
# the text
# You can get the shape after you transform


# transform
vectorizer.transform(sample_text_2)


# check the shape
len(vectorizer.vocabulary_)
```

```
{'one': 2, 'we': 4, 'can': 0, 'dance': 1, 'such': 3}
```

Out[13]:
```
5
```

In [14]:
```
# Now, we can inspect how our vectorizer vectorized the text
# This will print out a list of words used, and their index in the
# vectors. As you notice, all the vocabulary is lower-cased!! Also,
# the punctuation is left out. Single letter words is also eliminated.

# see the vocabulary created for the data set
print(vectorizer.vocabulary_)
```

```
{'one': 2, 'we': 4, 'can': 0, 'dance': 1, 'such': 3}
```

In [24]:
```
# If we would like to actually create a vector, we can do so by
# passing the text into the vectorizer to get back counts
# sparse matrix -> dense matrix (you can use toarray() or to_dense())
vector = vectorizer.transform(sample_text_2)
print(vector.shape)
print(vector)
print(vector.toarray())
print(vector.todense())
```

```
(1, 5)
  (0, 0)	1
  (0, 1)	2
  (0, 2)	3
  (0, 3)	1
  (0, 4)	1
[[1 2 3 1 1]]
[[1 2 3 1 1]]
```

In [20]:
```
# Or if we wanted to get the vector for one word:
print(vectorizer.transform(['one']).toarray())
# OOV (out-of-vocabulary)

print(vectorizer.transform(['hot']).toarray())
```

```
[[0 0 1 0 0]]
[[0 0 0 0 0]]
```

In [ ]:
```
# That is how you get the vector for test data.
```

In [21]:
```
# Let's repeat the example with another text
new_text = [' Today is the day that I do the thing today, today']

#instantiate the CountVectorizer using a user-defined tokenizer
new_vectorizer = CountVectorizer()
```

In [22]:
```
# print the vector after fit_transform
new_vectorizer.fit_transform(new_text)

# look at the vocabulary
new_vectorizer.vocabulary_
```

Out[22]:
```
{'today': 6, 'is': 2, 'the': 4, 'day': 0, 'that': 3, 'do': 1, 'thing': 5}
```

In [23]:
```
# what is the shape of the vector

new_vectorizer.transform(new_text).shape
```

Out[23]:
```
(1, 7)
```

TfidfVectorizer, TfidfTransformer

The TfidfVectorizer will tokenize documents, learn the vocabulary and inverse document frequency weights, and allow you to encode new documents. Alternately, if you already have a learned CountVectorizer, you can use it with a TfidfTransformer to just calculate the inverse document frequencies and start encoding documents.

The same create, fit, and transform process is used as with the CountVectorizer.

In [26]:
```
# import the TfidfVectorizer, and TfidfTransformer
from sklearn.feature_extraction.text import TfidfVectorizer

# instantiate the vectorizer
vectorizer = TfidfVectorizer()
```

In [27]:
```
# fit the vectorizer for sample_text_2
vectorizer.fit(sample_text_2)
```
Out[27]:
```
TfidfVectorizer()
```

In [44]:
```
# observe the vocabulary
vectorizer.vocabulary_
```
Out[44]:
```
{'one': 2, 'we': 4, 'can': 0, 'dance': 1, 'such': 3}
```

In [45]:
```
# transform the vectorizer

vector = vectorizer.transform(sample_text_2)
```

In [54]:
```
# shape of the vector
print(vector.shape)
print(vector.todense())
```
```
(1, 5)
[[0.25 0.5  0.75 0.25 0.25]]
```

In [47]:
```
# if you want to see your vector, convert it to dense array
print(vector.todense())
```
```
[[0.25 0.5  0.75 0.25 0.25]]
```

In [57]:
```
from sklearn.feature_extraction.text import TfidfTransformer
#instantiate CountVectorizer()
cv=CountVectorizer()

# this steps generates word counts for the words in your docs
word_count_vector=cv.fit_transform(sample_text_2)
print(word_count_vector.todense())
print(cv.vocabulary_)
tfidf_transformer=TfidfTransformer()
tfidf_transformer.fit(word_count_vector)
print(tfidf_transformer.idf_)
tfidf_transformer.transform(word_count_vector).todense()
```
```
[[1 2 3 1 1]]
{'one': 2, 'we': 4, 'can': 0, 'dance': 1, 'such': 3}
[1. 1. 1. 1. 1.]
```
Out[57]:
```
matrix([[0.25, 0.5 , 0.75, 0.25, 0.25]])
```

In [ ]:
```
# alternatively, you can use toarray()
```

In [58]:
```
# let's see another text
text = ["The quick brown fox jumped over the lazy dog.",
        "The dog.",
        "The fox"]
```

In [59]:
```
# fit the vectorizer
vectorizer.fit(text)
```

Out[59]:
```
TfidfVectorizer()
```

In [60]:
```
# observe the vocabulary

vectorizer.vocabulary_
```

Out[60]:
```
{'the': 7,
 'quick': 6,
 'brown': 0,
 'fox': 2,
 'jumped': 3,
 'over': 5,
 'lazy': 4,
 'dog': 1}
```

In [61]:
```
# you can see the idf values with .idf_

vectorizer.idf_
```

Out[61]:
```
array([1.69314718, 1.28768207, 1.28768207, 1.69314718, 1.69314718,
       1.69314718, 1.69314718, 1.        ])
```

In [65]:
```
# transform the vectorizer for the first document.
print(text[0])
vector=vectorizer.transform([text[0]])
```

```
The quick brown fox jumped over the lazy dog.
```

In [66]:
```
# shape of the vector
vector.shape
```

Out[66]:
```
(1, 8)
```

In [67]:
```
# look at the vector by converting to dense array
vector.toarray()
```

Out[67]:
```
array([[0.36388646, 0.27674503, 0.27674503, 0.36388646, 0.36388646,
        0.36388646, 0.36388646, 0.42983441]])
```

In [73]:
```
# let's see the vector in a df format
# vectorizer.get_feature_names() returns your features
import pandas as pd
vectorizer = TfidfVectorizer()
vector = vectorizer.fit_transform(text)
feature_name = vectorizer.get_feature_names()

dense = vector.todense()
denselist = dense.tolist()
df = pd.DataFrame(denselist, columns=feature_name)
```

In [74]:
```
# look at your dataframe
df
```

Out[74]:

	brown	dog	fox	jumped	lazy	over	quick	the
0	0.363886	0.276745	0.276745	0.363886	0.363886	0.363886	0.363886	0.429834
1	0.000000	0.789807	0.000000	0.000000	0.000000	0.000000	0.000000	0.613356
2	0.000000	0.000000	0.789807	0.000000	0.000000	0.000000	0.000000	0.613356

In [ ]:
```
```

min_df

CountVectorizer tokenizes using a regular expression "\b\w\w+\b". It also converts all to lowercase letters.

Note: \w\w+ translates to [a-zA-Z0-9][a-zA-Z0-9]+ (which can be written as [a-zA-Z0-9_]{2,}). This matches 2 or more alphanumeric characters (as defined between the square brackets).

The \b matches word boundaries: anything that is not an alphanumeric character, next to something alphanumeric. This includes spaces and punctuation, so it also includes the dot and causes the separation.

Stop words

Fixed sets of stopwords are not very likely to increase the performance. Fixed sets are usually good for small datasets. So, let's try another approach. Removing stopwords might not always be a good idea especially in sentiment analysis. By removing the stop word, a negative rating could turn out to be positive and vice versa.

HashingVectorizer

Note that this vectorizer does not require a call to fit on the training data documents. Instead, after instantiation, it can be used directly to start encoding documents.

The values of the encoded document correspond to normalized word counts by default in the range of -1

In [127…
```
from sklearn.feature_extraction.text import HashingVectorizer
# list of text docs
sample = ["The quick brown fox jumped over the lazy dog."]
```

In [128…
```
# create the transform with 5 features

hvectorizer = HashingVectorizer(n_features = 10, norm=None, alternate_sign=Fa
```

In [129…
```
# encode the doc
hvector = hvectorizer.fit_transform(sample)

#print(vector.vocabulary_)
```

In [130…
```
sample
```

Out[130…
```
['The quick brown fox jumped over the lazy dog.']
```

In [131…
```
# print the shape of your vector

#print(dir(vector))
hvector.shape
```

Out[131…
```
(1, 10)
```

In [132…
```
hvector.todense()
```

Out[132…
```
matrix([[0., 1., 0., 2., 0., 2., 0., 1., 3., 0.]])
```

In [133…
```
# print populated columns of first document
# format: (doc id, pos_in_matrix)  raw_count
print(hvector[0])
```

```
  (0, 1)	1.0
  (0, 3)	2.0
  (0, 5)	2.0
  (0, 7)	1.0
  (0, 8)	3.0
```

In [126…
```
from sklearn.feature_extraction.text import CountVectorizer
cvectorizer = CountVectorizer()

# compute counts without any term frequency normalization
X = cvectorizer.fit_transform(sample)

# print populated columns of first document
# format: (doc id, pos_in_matrix)  raw_count
print(X[0])
```

```
  (0, 7)	2
  (0, 6)	1
  (0, 0)	1
  (0, 2)	1
  (0, 3)	1
  (0, 5)	1
  (0, 4)	1
  (0, 1)	1
```

In [136…

```
hvectorizer = HashingVectorizer(n_features=100,norm=None,alternate_sign=False

# compute counts without any term frequency normalization
X = hvectorizer.fit_transform(sample)
# print populated columns of first document
# format: (doc id, pos_in_matrix)  raw_count
print(X[0])
```

```
  (0, 11)	1.0
  (0, 15)	1.0
  (0, 53)	1.0
  (0, 58)	2.0
  (0, 85)	1.0
  (0, 87)	1.0
  (0, 88)	1.0
  (0, 93)	1.0
```

In [ ]: