10.2 The Vector Space Model

May 9, 2019

1 10.2 The Vector Space Model

In the previous section, we learned how to convert a document into a bag of words (or, more generally, a bag of n-grams) representation. In this section, we go one step further: how to turn the bag of words representation into the rows of a `DataFrame`.

Before we dive into the details, the representation of a document by a vector of numbers is called the **vector space model**. There are many ways to convert a bag of words representation into a vector of numbers, some of which we explore in this section.

```
In [1]: import numpy as np
        import pandas as pd
        pd.options.display.max_rows = 10
        from collections import Counter

        sms = pd.read_csv(
            "https://raw.githubusercontent.com/dlsun/data-science-book/master/data/SMSSpamColle
            sep="\t",
            names=["label", "text"]
        )
```

1.1 Term Frequencies

The bag of words representation gives us a list of word counts, like `{"I": 2, "am": 2, "Sam": 2}`. To turn this into a vector of numbers, we can simply take the word counts, for each word in a prespecified vocabulary, as follows:

a	I	am	the	Sam	...
0	2	2	0	2	...

We can do this for each document in the corpus, to obtain the **term-frequency matrix**.

Let's obtain the term-frequency matrix for the text message corpus. But let's restrict to just the first 100 messages and just words containing only letters. (Otherwise, we end up with "words" that are phone numbers and addresses.)

```
In [2]: from collections import Counter
```

```
bag_of_words = (
    sms.loc[:100, "text"].
    str.lower().
    str.replace("[^A-Za-z\s]", "").
    str.split()
).apply(Counter)

bag_of_words
```

```
Out[2]: 0      {'go': 1, 'until': 1, 'jurong': 1, 'point': 1,...
        1      {'ok': 1, 'lar': 1, 'joking': 1, 'wif': 1, 'u'...
        2      {'free': 1, 'entry': 2, 'in': 1, 'a': 1, 'wkly...
        3      {'u': 2, 'dun': 1, 'say': 2, 'so': 1, 'early':...
        4      {'nah': 1, 'i': 1, 'dont': 1, 'think': 1, 'he'...
                                     ...
        96     {'watching': 1, 'telugu': 1, 'moviewat': 1, 'a...
        97     {'i': 1, 'see': 1, 'when': 1, 'we': 2, 'finish...
        98     {'hi': 1, 'wk': 1, 'been': 1, 'ok': 1, 'on': 2...
        99     {'i': 1, 'see': 1, 'a': 1, 'cup': 1, 'of': 1, ...
        100    {'please': 1, 'dont': 1, 'text': 1, 'me': 1, '...
        Name: text, Length: 101, dtype: object
```

To make a term-frequency matrix out of this data, we need to convert it to a `DataFrame`, where each column represents a word and each row a document—and the cells contain the count of that word in the document.

```
In [3]: tf = pd.DataFrame(list(bag_of_words))
        tf
```

```
Out[3]:        a  abiola  about  abt   ac  accomodations  acoentry  actin  advise  aft  \
        0    NaN     NaN    NaN  NaN  NaN            NaN       NaN    NaN     NaN  NaN
        1    NaN     NaN    NaN  NaN  NaN            NaN       NaN    NaN     NaN  NaN
        2    1.0     NaN    NaN  NaN  NaN            NaN       NaN    NaN     NaN  NaN
        3    NaN     NaN    NaN  NaN  NaN            NaN       NaN    NaN     NaN  NaN
        4    NaN     NaN    NaN  NaN  NaN            NaN       NaN    NaN     NaN  NaN
        ..   ...     ...    ...  ...   ..            ...       ...    ...     ...  ...
        96   NaN     NaN    NaN  1.0  NaN            NaN       NaN    NaN     NaN  NaN
        97   NaN     NaN    NaN  NaN  NaN            NaN       NaN    NaN     NaN  NaN
        98   2.0     NaN    NaN  NaN  NaN            NaN       NaN    NaN     NaN  NaN
        99   1.0     NaN    NaN  NaN  NaN            NaN       NaN    NaN     NaN  NaN
        100  NaN     NaN    NaN  NaN  NaN            NaN       NaN    NaN     NaN  NaN

             ...   yo  you  youd  youhow  youll  your  yours  yourself  yummy  yup
        0    ...  NaN  NaN   NaN     NaN    NaN   NaN    NaN       NaN    NaN  NaN
        1    ...  NaN  NaN   NaN     NaN    NaN   NaN    NaN       NaN    NaN  NaN
        2    ...  NaN  NaN   NaN     NaN    NaN   NaN    NaN       NaN    NaN  NaN
        3    ...  NaN  NaN   NaN     NaN    NaN   NaN    NaN       NaN    NaN  NaN
        4    ...  NaN  NaN   NaN     NaN    NaN   NaN    NaN       NaN    NaN  NaN
        ..   ...   ..  ...   ...     ...    ...   ...    ...       ...    ...  ...
```

```
96  ...  NaN  NaN   NaN     NaN    NaN   NaN    NaN       NaN    NaN  NaN
97  ...  NaN  NaN   NaN     NaN    NaN   NaN    NaN       NaN    NaN  NaN
98  ...  NaN  NaN   NaN     NaN    NaN   NaN    NaN       NaN    NaN  NaN
99  ...  NaN  NaN   NaN     NaN    NaN   NaN    NaN       NaN    NaN  NaN
100 ...  NaN  NaN   NaN     NaN    NaN   NaN    NaN       NaN    NaN  NaN

[101 rows x 707 columns]
```

Although there are a few numbers in this matrix, it is mostly NaNs. That simply means that the word did not appear in the dictionary for that document. In other words, a NaN really means a count of 0. So let's replace the NaNs by 0s.

```
In [4]: tf = tf.fillna(0)
        tf
```

```
Out[4]:       a  abiola  about  abt   ac  accomodations  acoentry  actin  advise  \
        0    0.0     0.0    0.0  0.0  0.0            0.0       0.0    0.0     0.0
        1    0.0     0.0    0.0  0.0  0.0            0.0       0.0    0.0     0.0
        2    1.0     0.0    0.0  0.0  0.0            0.0       0.0    0.0     0.0
        3    0.0     0.0    0.0  0.0  0.0            0.0       0.0    0.0     0.0
        4    0.0     0.0    0.0  0.0  0.0            0.0       0.0    0.0     0.0
        ..   ...     ...    ...  ...  ...            ...       ...    ...     ...
        96   0.0     0.0    0.0  1.0  0.0            0.0       0.0    0.0     0.0
        97   0.0     0.0    0.0  0.0  0.0            0.0       0.0    0.0     0.0
        98   2.0     0.0    0.0  0.0  0.0            0.0       0.0    0.0     0.0
        99   1.0     0.0    0.0  0.0  0.0            0.0       0.0    0.0     0.0
        100  0.0     0.0    0.0  0.0  0.0            0.0       0.0    0.0     0.0

             aft ...   yo  you  youd  youhow  youll  your  yours  yourself  yummy  \
        0    0.0 ...  0.0  0.0   0.0     0.0    0.0   0.0    0.0       0.0    0.0
        1    0.0 ...  0.0  0.0   0.0     0.0    0.0   0.0    0.0       0.0    0.0
        2    0.0 ...  0.0  0.0   0.0     0.0    0.0   0.0    0.0       0.0    0.0
        3    0.0 ...  0.0  0.0   0.0     0.0    0.0   0.0    0.0       0.0    0.0
        4    0.0 ...  0.0  0.0   0.0     0.0    0.0   0.0    0.0       0.0    0.0
        ..   ... ...  ...  ...   ...     ...    ...   ...    ...       ...    ...
        96   0.0 ...  0.0  0.0   0.0     0.0    0.0   0.0    0.0       0.0    0.0
        97   0.0 ...  0.0  0.0   0.0     0.0    0.0   0.0    0.0       0.0    0.0
        98   0.0 ...  0.0  0.0   0.0     0.0    0.0   0.0    0.0       0.0    0.0
        99   0.0 ...  0.0  0.0   0.0     0.0    0.0   0.0    0.0       0.0    0.0
        100  0.0 ...  0.0  0.0   0.0     0.0    0.0   0.0    0.0       0.0    0.0

             yup
        0    0.0
        1    0.0
        2    0.0
        3    0.0
        4    0.0
        ..   ...
```

```
96   0.0
97   0.0
98   0.0
99   0.0
100  0.0

[101 rows x 707 columns]
```

You might be tempted at this point to run the same code on the entire corpus of text messages. But the number of columns (i.e., the size of the vocabulary) quickly grows out of control. There are about 9000 unique words in the entire corpus, and storing that many columns is on the edge of what pandas can handle.

But we observed above that *most of the entries in this matrix are zero.* Instead of storing all the entries in this matrix, we can simply store the locations (row and column index) of the non-zero elements and their values. All of the remaining entries are assumed to be zeroes. This is called a **sparse** representation of the matrix.

To get a sparse representation of the term-frequency matrix, we use the `CountVectorizer` object in Scikit-Learn. This object takes in a list of strings, splits each string into words, counts them, and returns the term-frequency matrix. By default, it converts all letters to lowercase and strips punctuation, although this behavior can be customized.

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

        vec = CountVectorizer()
        vec.fit(sms["text"]) # This determines the vocabulary.
        tf_sparse = vec.transform(sms["text"])
```

A sparse matrix can be converted to a **dense** matrix if necessary, using the `.todense()` method. But be careful. If the matrix is large, you do not want to do this!

```
In [6]: tf_sparse.todense()

Out[6]: matrix([[0, 0, 0, ..., 0, 0, 0],
                [0, 0, 0, ..., 0, 0, 0],
                [0, 0, 0, ..., 0, 0, 0],
                ...,
                [0, 0, 0, ..., 0, 0, 0],
                [0, 0, 0, ..., 0, 0, 0],
                [0, 0, 0, ..., 0, 0, 0]])
```

Notice that the resulting object is no longer a `DataFrame`. It is simply a matrix of numbers. Each column corresponds to a word (and, if necessary, we can find the mapping between words and columns in `vec.vocabulary_`). But the word counts themselves are not of primary interest. We now have a completely numerical representation of every text document that can be passed into a machine learning model, like k-nearest neighbors.

We can even count bigrams using `CountVectorizer` by specifying `ngram_range`. If we wanted both unigrams (i.e., individual words) and the bigrams, then we would specify `ngram_range=(1, 2)`. If we want just the bigrams, then we would specify `ngram_range=(2, 2)`. Let's do the latter: