Lecture 8
Textual Data: Bag-of-Words and N-Grams

April 16, 2025

Roadmap for Today

Many data science techniques assume that all the variables are quantitative.

Roadmap for Today

Many data science techniques assume that all the variables are quantitative.

- *Example:* measuring similarity / calculating distances between observations

Roadmap for Today

Many data science techniques assume that all the variables are quantitative.

- *Example:* measuring similarity / calculating distances between observations

Last time, we learned how to convert categorical variables to quantitative variables.

Roadmap for Today

Many data science techniques assume that all the variables are quantitative.

- *Example:* measuring similarity / calculating distances between observations

Last time, we learned how to convert categorical variables to quantitative variables.

Today, we will learn how to convert a completely new type of data to quantitative variables.

Textual Data

A textual data set consists of multiple texts. Each text is called a **document**. The collection of texts is called a **corpus**.

Textual Data

A textual data set consists of multiple texts. Each text is called a **document**. The collection of texts is called a **corpus**.

Example Corpus:

1. "I am Sam\n\nI am Sam\nSam I..."
2. "The sun did not shine.\nIt was..."
3. "Fox\nSocks\nBox\nKnox\n\nKnox..."
4. "Every Who\nDown in Whoville\n..."
5. "UP PUP Pup is up.\nCUP PUP..."
6. "On the fifteenth of May, in the..."
7. "Congratulations!\nToday is your..."
8. "One fish, two fish, red fish..."

Reading in Textual Data

Documents are usually stored in different files.

```
seuss_dir = "http://dlsun.github.io/pods/data/drseuss/"
seuss_files = [
    "green_eggs_and_ham.txt", "cat_in_the_hat.txt",
    "fox_in_socks.txt", "how_the_grinch_stole_christmas.txt",
    "hop_on_pop.txt", "horton_hears_a_who.txt",
    "oh_the_places_youll_go.txt", "one_fish_two_fish.txt"]
```

seuss_dir → Stores the web directory where the Dr. Seuss books are located.

seuss_files → A list containing the filenames of the Dr. Seuss books in that directory.

Reading in Textual Data

Documents are usually stored in different files.

```
seuss_dir = "http://dlsun.github.io/pods/data/drseuss/"
seuss_files = [
    "green_eggs_and_ham.txt", "cat_in_the_hat.txt",
    "fox_in_socks.txt", "how_the_grinch_stole_christmas.txt",
    "hop_on_pop.txt", "horton_hears_a_who.txt",
    "oh_the_places_youll_go.txt", "one_fish_two_fish.txt"]
```

We have to read them in one by one.

Reading in Textual Data

Documents are usually stored in different files.

```
seuss_dir = "http://dlsun.github.io/pods/data/drseuss/"
seuss_files = [
    "green_eggs_and_ham.txt", "cat_in_the_hat.txt",
    "fox_in_socks.txt", "how_the_grinch_stole_christmas.txt",
    "hop_on_pop.txt", "horton_hears_a_who.txt",
    "oh_the_places_youll_go.txt", "one_fish_two_fish.txt"]
```

We have to read them in one by one.

```
import requests

docs = {}
for filename in seuss_files:
    response = requests.get(seuss_dir + filename, "r")
    docs[filename] = response.text
```

Sends a request to download each file from the Dr. Seuss directory
Reads the text from each file
Stores each book's text in a dictionary called **docs,** using the filename as the key

Textual Data

A textual data set consists of several texts. Each text is called a **document**. The collection of texts is called a **corpus**.

Example Corpus:

1. "I am Sam\n\nI am Sam\nSam I..."
2. "The sun did not shine.\nIt was..."
3. "Fox\nSocks\nBox\nKnox\n\nKnox..."
4. "Every Who\nDown in Whoville\n..."
5. "UP PUP Pup is up.\nCUP PUP..."
6. "On the fifteenth of May, in the..."
7. "Congratulations!\nToday is your..."
8. "One fish, two fish, red fish..."

Textual Data

A textual data set consists of several texts. Each text is called a **document**. The collection of texts is called a **corpus**.

Example Corpus:

1. "I am Sam\n\nI am Sam\nSam I..."
2. "The sun did not shine.\nIt was..."
3. "Fox\nSocks\nBox\nKnox\n\nKnox..."
4. "Every Who\nDown in Whoville\n..." →
5. "UP PUP Pup is up.\nCUP PUP..."
6. "On the fifteenth of May, in the..."
7. "Congratulations!\nToday is your..."
8. "One fish, two fish, red fish..."

Goal: Turn this corpus into a matrix of numbers.

Textual Data

A textual data set consists of several texts. Each text is called a **document**. The collection of texts is called a **corpus**.

Example Corpus:

0 ❶ "I am Sam\n\nI am Sam\nSam I..."

1 ❷ "The sun did not shine.\nIt was..."

2 ❸ "Fox\nSocks\nBox\nKnox\n\nKnox..."

3 ❹ "Every Who\nDown in Whoville\n..."

4 ❺ "UP PUP Pup is up.\nCUP PUP..."

5 ❻ "On the fifteenth of May, in the..."

6 ❼ "Congratulations!\nToday is your..."

7 ❼ "One fish, two fish, red fish..."

→

0	1	0	2	...
1	0	1	0	...
2	3	0	0	...
3	0	2	1	...
4	0	0	1	...
5	2	0	5	...
6	0	0	0	...
7	0	2	0	...

Goal: Turn this corpus into a matrix of numbers.

Textual Data

A textual data set consists of several texts. Each text is called a **document**. The collection of texts is called a **corpus**.

Example Corpus:

1. "I am Sam\n\nI am Sam\nSam I..."
2. "The sun did not shine.\nIt was..."
3. "Fox\nSocks\nBox\nKnox\n\nKnox..."
4. "Every Who\nDown in Whoville\n..."
5. "UP PUP Pup is up.\nCUP PUP..."
6. "On the fifteenth of May, in the..."
7. "Congratulations!\nToday is your..."
8. "One fish, two fish, red fish..."

→

	?	?	?	
0	1	0	2	...
1	0	1	0	...
2	3	0	0	...
3	0	2	1	...
4	0	0	1	...
5	2	0	5	...
6	0	0	0	...
7	0	2	0	...

Goal: Turn this corpus into a matrix of numbers.

But what would each column represent?!

Bag-of-Words Model

In the **bag-of-words model**, each column represents a word, and the values in the column are the word counts.

Bag-of-Words Model

In the **bag-of-words model**, each column represents a word, and the values in the column are the word counts.

First, we need to count the words in each document.

```
from collections import Counter
Counter(docs["hop_on_pop.txt"].split())
```

1. Imports Counter, a Python tool used for counting occurrences of items.
2. Accesses the text of the document from the docs dictionary.
3. Splits the document text into individual words.
4. Counts how many times each word appears in the document.
Results:
Produces a word frequency dictionary
e.g. Counter({'I': 5, 'am': 3, 'Sam': 2, ...})

text = "I am Sam I am"
Counter(text.split())

Counter({'I': 2, 'am': 2, 'Sam': 1})

Bag-of-Words Model

In the **bag-of-words model**, each column represents a word, and the values in the column are the word counts.

First, we need to count the words in each document.

```
from collections import Counter
Counter(docs["hop_on_pop.txt"].split())
```

```
Counter({'UP': 1, 'PUP': 3, 'Pup': 4, 'is': 10, 'up.': 2, ...})
```

Bag-of-Words Model

In the **bag-of-words model**, each column represents a word, and the values in the column are the word counts.

First, we need to count the words in each document.

```
from collections import Counter
Counter(docs["hop_on_pop.txt"].split())
```

```
Counter({'UP': 1, 'PUP': 3, 'Pup': 4, 'is': 10, 'up.': 2, ...})
```

We stack these counts into a `DataFrame`.

Bag-of-Words Model

In the **bag-of-words model**, each column represents a word, and the values in the column are the word counts.

First, we need to count the words in each document.

```
from collections import Counter
Counter(docs["hop_on_pop.txt"].split())
```

```
Counter({'UP': 1, 'PUP': 3, 'Pup': 4, 'is': 10, 'up.': 2, ...})
```

We stack these counts into a DataFrame.

```
import pandas as pd
pd.DataFrame(
    [Counter(doc.split()) for doc in docs.values()],
    index=docs.keys())
```

key = file name (book title), value = book text

Counter(doc.split()) for doc in docs.values()
docs.values() → get all the book textsdoc.
split() → split the text into words (separate by spaces)
Counter(...) → count how many times each word appears

index=docs.keys()
It sets the DataFrame row labels to the file names, meaning each row represents one book.

Bag-of-Words Model

In the **bag-of-words model**, each column represents a word, and the values in the column are the word counts.

First, we need to count the words in each document.

```
from collections import Counter
Counter(docs["hop_on_pop.txt"].split())
```

```
Counter({'UP': 1, 'PUP': 3, 'Pup': 4, 'is': 10, 'up.': 2, ...})
```

We stack these counts into a DataFrame.

```
import pandas as pd
pd.DataFrame(
    [Counter(doc.split()) for doc in docs.values()],
    index=docs.keys())
```

	I	am	Sam	That	Sam-I-am	Sam-I-am!	do	not	like	that	...
green_eggs_and_ham.txt	71.0	3.0	3.0	2.0	4.0	2.0	34.0	46.0	44.0	1.0	...
cat_in_the_hat.txt	48.0	NaN	NaN	4.0	NaN	NaN	13.0	27.0	13.0	16.0	...
fox_in_socks.txt	9.0	NaN	NaN	NaN	NaN	NaN	6.0	1.0	1.0	1.0	...
hop_on_pop.txt	2.0	1.0	NaN	2.0	NaN	NaN	NaN	2.0	5.0	2.0	...
horton_hears_a_who.txt	18.0	1.0	NaN	7.0	NaN	NaN	NaN	3.0	NaN	24.0	...
how_the_grinch_stole_christmas.txt	6.0	NaN	NaN	2.0	NaN	NaN	2.0	1.0	2.0	11.0	...
oh_the_places_youll_go.txt	2.0	NaN	NaN	NaN	NaN	NaN	2.0	6.0	1.0	11.0	...
one_fish_two_fish.txt	48.0	3.0	NaN	NaN	NaN	NaN	11.0	9.0	21.0	1.0	...

8 rows × 2562 columns

Bag-of-Words Model

In the **bag-of-words model**, each column represents a word, and the values in the column are the word counts.

First, we need to count the words in each document.

```
from collections import Counter
Counter(docs["hop_on_pop.txt"].split())
```

```
Counter({'UP': 1, 'PUP': 3, 'Pup': 4, 'is': 10, 'up.': 2, ...})
```

We stack these counts into a `DataFrame`.

```
import pandas as pd
pd.DataFrame(
    [Counter(doc.split()) for doc in docs.values()],
    index=docs.keys())
```

	I	am	Sam	That	Sam-I-am	Sam-I-am!	do	not	like	that	...
green_eggs_and_ham.txt	71.0	3.0	3.0	2.0	4.0	2.0	34.0	46.0	44.0	1.0	...
cat_in_the_hat.txt	48.0	NaN	NaN	4.0	NaN	NaN	13.0	27.0	13.0	16.0	...
fox_in_socks.txt	9.0	NaN	NaN	NaN	NaN	NaN	6.0	1.0	1.0	1.0	...
hop_on_pop.txt	2.0	1.0	NaN	2.0	NaN	NaN	NaN	2.0	5.0	2.0	...
horton_hears_a_who.txt	18.0	1.0	NaN	7.0	NaN	NaN	NaN	3.0	NaN	24.0	...
how_the_grinch_stole_christmas.txt	6.0	NaN	NaN	2.0	NaN	NaN	2.0	1.0	2.0	11.0	...
oh_the_places_youll_go.txt	2.0	NaN	NaN	NaN	NaN	NaN	2.0	6.0	1.0	11.0	...
one_fish_two_fish.txt	48.0	3.0	NaN	NaN	NaN	NaN	11.0	9.0	21.0	1.0	...

8 rows × 2562 columns

To get rid of the NaNs, add `.fillna(0)`.

Bag-of-Words Model

In the **bag-of-words model**, each column represents a word, and the values in the column are the word counts.

First, we need to count the words in each document.

```
from collections import Counter
Counter(docs["hop_on_pop.txt"].split())
```

```
Counter({'UP': 1, 'PUP': 3, 'Pup': 4, 'is': 10, 'up.': 2, ...})
```

We stack these counts into a `DataFrame`.

```
import pandas as pd
pd.DataFrame(
    [Counter(doc.split()) for doc in docs.values()],
    index=docs.keys())
```

	I	am	Sam	That	Sam-I-am	Sam-I-am!	do	not	like	that	...
green_eggs_and_ham.txt	71.0	3.0	3.0	2.0	4.0	2.0	34.0	46.0	44.0	1.0	...
cat_in_the_hat.txt	48.0	NaN	NaN	4.0	NaN	NaN	13.0	27.0	13.0	16.0	...
fox_in_socks.txt	9.0	NaN	NaN	NaN	NaN	NaN	6.0	1.0	1.0	1.0	...
hop_on_pop.txt	2.0	1.0	NaN	2.0	NaN	NaN	NaN	2.0	5.0	2.0	...
horton_hears_a_who.txt	18.0	1.0	NaN	7.0	NaN	NaN	NaN	3.0	NaN	24.0	...
how_the_grinch_stole_christmas.txt	6.0	NaN	NaN	2.0	NaN	NaN	2.0	1.0	2.0	11.0	...
oh_the_places_youll_go.txt	2.0	NaN	NaN	NaN	NaN	NaN	2.0	6.0	1.0	11.0	...
one_fish_two_fish.txt	48.0	3.0	NaN	NaN	NaN	NaN	11.0	9.0	21.0	1.0	...

8 rows × 2562 columns

To get rid of the NaNs, add `.fillna(0)`.

This is called the **term-frequency matrix**.

Bag-of-Words in Scikit-Learn

Alternatively, we can use `CountVectorizer` in scikit-learn to produce a term-frequency matrix.

Bag-of-Words in Scikit-Learn

Alternatively, we can use `CountVectorizer` in scikit-learn to produce a term-frequency matrix.

```
from sklearn.feature_extraction.text import CountVectorizer
vec = CountVectorizer()
vec.fit(docs.values())
vec.transform(docs.values())
```

1. Imports a tool(classl that converts text into a matrix of word

2. Creates a CountVectorizer object.

3. Builds the vocabulary by looking at all the words in the documents.

4. Converts each document into a numeric vector based on how many times each vocabulary word appears.

You get a **document-term matrix**:
Rows → documents
Columns → unique words (vocabulary)
Values → word counts

This is the automated version of what Counter() does manually.

Bag-of-Words in Scikit-Learn

Alternatively, we can use `CountVectorizer` in scikit-learn to produce a term-frequency matrix.

```
from sklearn.feature_extraction.text import CountVectorizer
vec = CountVectorizer()
vec.fit(docs.values())
vec.transform(docs.values())
```

```
<8x1344 sparse matrix of type '<class 'numpy.int64'>'
        with 2308 stored elements in Compressed Sparse Row format>
```

Bag-of-Words in Scikit-Learn

Alternatively, we can use CountVectorizer in scikit-learn to produce a term-frequency matrix.

```
from sklearn.feature_extraction.text import CountVectorizer
vec = CountVectorizer()
vec.fit(docs.values())
vec.transform(docs.values())
```

```
<8x1344 sparse matrix of type '<class 'numpy.int64'>'
	with 2308 stored elements in Compressed Sparse Row format>
```

- The set of words across a corpus is called the **vocabulary**. We can view the vocabulary in a fitted CountVectorizer as follows:

```
vec.vocabulary_
```

prints something like

Bag-of-Words in Scikit-Learn

Alternatively, we can use CountVectorizer in scikit-learn to produce a term-frequency matrix.

```
from sklearn.feature_extraction.text import CountVectorizer
vec = CountVectorizer()
vec.fit(docs.values())
vec.transform(docs.values())
```

```
<8x1344 sparse matrix of type '<class 'numpy.int64'>'
	with 2308 stored elements in Compressed Sparse Row format>
```

The set of words across a corpus is called the **vocabulary**. We can view the vocabulary in a fitted CountVectorizer as follows:

```
vec.vocabulary_
```

```
{'am': 23, 'sam': 935, 'that': 1138, 'do': 287, 'not': 767, ...}
```

Bag-of-Words in Scikit-Learn

Alternatively, we can use CountVectorizer in scikit-learn to produce a term-frequency matrix.

```
from sklearn.feature_extraction.text import CountVectorizer
vec = CountVectorizer()
vec.fit(docs.values())
vec.transform(docs.values())
```

```
<8x1344 sparse matrix of type '<class 'numpy.int64'>'
	with 2308 stored elements in Compressed Sparse Row format>
```

The set of words across a corpus is called the **vocabulary**. We can view the vocabulary in a fitted CountVectorizer as follows:

```
vec.vocabulary_
```

```
{'am': 23, 'sam': 935, 'that': 1138, 'do': 287, 'not': 767, ...}
```

The number here represents the column index in the matrix! (So column 23 contains the counts for "am", etc.)

Bag-of-Words in Scikit-Learn

Alternatively, we can use `CountVectorizer` in scikit-learn to produce a term-frequency matrix.

```
from sklearn.feature_extraction.text import CountVectorizer
vec = CountVectorizer()
vec.fit(docs.values())
vec.transform(docs.values())
```

```
<8x1344 sparse matrix of type '<class 'numpy.int64'>'
	with 2308 stored elements in Compressed Sparse Row format>
```

Wait! Why are there only 1344 words?

The set of words across a corpus is called the **vocabulary**. We can view the vocabulary in a fitted `CountVectorizer` as follows:

Even if your total vocabulary is huge, CountVectorizer only counts **unique tokens** that actually appear in your dataset.

```
vec.vocabulary_
```

```
{'am': 23, 'sam': 935, 'that': 1138, 'do': 287, 'not': 767, ...}
```

The number here represents the column index in the matrix! (So column 23 contains the counts for "am", etc.)

Text Normalization

What's wrong with the way we counted words originally?

```
Counter({'UP': 1, 'PUP': 3, 'Pup': 4, 'is': 10, 'up.': 2, ...})
```

Text Normalization

What's wrong with the way we counted words originally?

```
Counter({'UP': 1, 'PUP': 3, 'Pup': 4, 'is': 10, 'up.': 2, ...})
```

It's usually good to **normalize** for punctuation and capitalization.

Text Normalization

What's wrong with the way we counted words originally?

`Counter({'UP': 1, 'PUP': 3, 'Pup': 4, 'is': 10, 'up.': 2, ...})`

It's usually good to **normalize** for punctuation and capitalization.

Normalization options are specified when you initialize the `CountVectorizer`. By default, Scikit-Learn strips punctuation and converts all characters to lowercase.

Text Normalization

What's wrong with the way we counted words originally?

`Counter({'UP': 1, 'PUP': 3, 'Pup': 4, 'is': 10, 'up.': 2, ...})`

It's usually good to **normalize** for punctuation and capitalization.

Normalization options are specified when you initialize the `CountVectorizer`. By default, Scikit-Learn strips punctuation and converts all characters to lowercase.

But if you don't want Scikit-Learn to normalize for punctuation and capitalization, you can do the following:

```
vec = CountVectorizer(lowercase=False, token_pattern=r"[\S]+")
vec.fit(docs.values())
vec.transform(docs.values())
```

lowercase=False →not convert words to lowercase.
It keeps the original casing of the text.

token_pattern=r"[\S]+" → treats anything separated by whitespace as a token (word)
r- raw string

Punctuation, numbers, and symbols are all counted as tokens
Example: "cat," is different from "cat"

Text Normalization

What's wrong with the way we counted words originally?

```
Counter({'UP': 1, 'PUP': 3, 'Pup': 4, 'is': 10, 'up.': 2, ...})
```

It's usually good to **normalize** for punctuation and capitalization.

Normalization options are specified when you initialize the `CountVectorizer`. By default, Scikit-Learn strips punctuation and converts all characters to lowercase.

But if you don't want Scikit-Learn to normalize for punctuation and capitalization, you can do the following:

```
vec = CountVectorizer(lowercase=False, token_pattern=r"[\S]+")
vec.fit(docs.values())
vec.transform(docs.values())
```

```
<8x2562 sparse matrix of type '<class 'numpy.int64'>'
	with 3679 stored elements in Compressed Sparse Row format>
```

Text Normalization

What's wrong with the way we counted words originally?

```
Counter({'UP': 1, 'PUP': 3, 'Pup': 4, 'is': 10, 'up.': 2, ...})
```

It's usually good to **normalize** for punctuation and capitalization.

Normalization options are specified when you initialize the `CountVectorizer`. By default, Scikit-Learn strips punctuation and converts all characters to lowercase.

But if you don't want Scikit-Learn to normalize for punctuation and capitalization, you can do the following:

```
vec = CountVectorizer(lowercase=False, token_pattern=r"[\S]+")
vec.fit(docs.values())
vec.transform(docs.values())
```

```
<8x2562 sparse matrix of type '<class 'numpy.int64'>'
        with 3679 stored elements in Compressed Sparse Row format>
```

Now we're back to 2562 words in the vocabulary!

1 Textual Data

2 Bag-of-Words Model

3 N-Grams

The Shortcomings of Bag-of-Words

Bag-of-words is easy to understand and easy to implement.

What are its disadvantages?

Bag-of-words is simple and useful, but it has several disadvantages:
1. Loses word order
It ignores grammar and sequence, so“dog bites man” = “man bites dog” → same representation.
2. Loses context and meaning
It doesn’t understand synonyms or context.“good” and “excellent” are treated as unrelated words.
3. High dimensionality
Every unique word becomes a feature, producing huge, sparse vectors.
4. Doesn’t handle unseen words well
New words in test data are ignored or cause issues.
5. No understanding of word importance
Common words (e.g., the, and) can dominate without additional techniques like TF-IDF.
6. Sensitive to vocabulary noise
Misspellings, punctuation, and capitalization create separate tokens unless preprocessing is strong.

The Shortcomings of Bag-of-Words

Bag-of-words is easy to understand and easy to implement.

What are its disadvantages?

Consider the following documents:

1. “The dog bit her owner.”
2. “Her dog bit the owner.”

The Shortcomings of Bag-of-Words

Bag-of-words is easy to understand and easy to implement.

What are its disadvantages?

Consider the following documents:

1. "The dog bit her owner."
2. "Her dog bit the owner."

Both documents have the same exact bag-of-words representation:

	the	her	dog	owner	bit
1	1	1	1	1	1
2	1	1	1	1	1

The Shortcomings of Bag-of-Words

Bag-of-words is easy to understand and easy to implement.

What are its disadvantages?

Consider the following documents:

1. “The dog bit her owner.”
2. “Her dog bit the owner.”

Both documents have the same exact bag-of-words representation:

	the	her	dog	owner	bit
1	1	1	1	1	1
2	1	1	1	1	1

But they mean something quite different!

N-grams

An **n-gram** is a sequence of *n* words.

N-grams

An **n-gram** is a sequence of *n* words.

Google Books Ngram Viewer

a tool that shows how often words or phrases appear in books over time, helping you see language and trend changes across years.

Uses Google Books data
Shows word/phrase frequency over time
Helps analyze language trends
Allows historical comparisons (e.g., “AI” vs “Machine learning”)
Displays results as a time-series graph

N-grams

An **n-gram** is a sequence of *n* words.

Google Books Ngram Viewer

N-grams allow us to capture more of the meaning.

N-grams

An **n-gram** is a sequence of *n* words.

Google Books Ngram Viewer

N-grams allow us to capture more of the meaning.

For example, if we count **bigrams** (2-grams) instead of words, we can distinguish the two documents from before:

1. “The dog bit her owner.”
2. “Her dog bit the owner.”

N-grams

An **n-gram** is a sequence of *n* words.

Google Books Ngram Viewer

N-grams allow us to capture more of the meaning.

For example, if we count **bigrams** (2-grams) instead of words, we can distinguish the two documents from before:

1. "The dog bit her owner."
2. "Her dog bit the owner."

	the,dog	her,dog	dog,bit	bit,the	bit,her	the,owner	her,owner
1	1	0	1	0	1	0	1
2	0	1	1	1	0	1	0

N-grams in Scikit-Learn

Scikit-Learn can create n-grams.

N-grams in Scikit-Learn

Scikit-Learn can create n-grams.

Just pass in `ngram_range=` to the `CountVectorizer`. To get bigrams, we set the range to `(2, 2)`:

```
vec = CountVectorizer(ngram_range=(2, 2))
vec.fit(docs.values())
vec.transform(docs.values())
```

N-grams in Scikit-Learn

Scikit-Learn can create n-grams.

Just pass in `ngram_range=` to the `CountVectorizer`. To get bigrams, we set the range to `(2, 2)`:

```
vec = CountVectorizer(ngram_range=(2, 2))
vec.fit(docs.values())
vec.transform(docs.values())
```

```
<8x5846 sparse matrix of type '<class 'numpy.int64'>'
        with 6459 stored elements in Compressed Sparse Row format>
```

N-grams in Scikit-Learn

Scikit-Learn can create n-grams.

Just pass in `ngram_range=` to the `CountVectorizer`. To get bigrams, we set the range to `(2, 2)`:

```
vec = CountVectorizer(ngram_range=(2, 2))
vec.fit(docs.values())
vec.transform(docs.values())
```

```
<8x5846 sparse matrix of type '<class 'numpy.int64'>'
        with 6459 stored elements in Compressed Sparse Row format>
```

We can also get individual words (unigrams) alongside the bigrams:

```
vec = CountVectorizer(ngram_range=(1, 2))
vec.fit(docs.values())
vec.transform(docs.values())
```

N-grams in Scikit-Learn

Scikit-Learn can create n-grams.

Just pass in `ngram_range=` to the `CountVectorizer`. To get bigrams, we set the range to `(2, 2)`:

```
vec = CountVectorizer(ngram_range=(2, 2))
vec.fit(docs.values())
vec.transform(docs.values())
```

```
<8x5846 sparse matrix of type '<class 'numpy.int64'>'
	with 6459 stored elements in Compressed Sparse Row format>
```

We can also get individual words (unigrams) alongside the bigrams:

```
vec = CountVectorizer(ngram_range=(1, 2))
vec.fit(docs.values())
vec.transform(docs.values())
```

```
<8x7190 sparse matrix of type '<class 'numpy.int64'>'
	with 8767 stored elements in Compressed Sparse Row format>
```