Intro to text mining - Text processing - 3

One should look for what is and not what he thinks should be. (Albert Einstein)

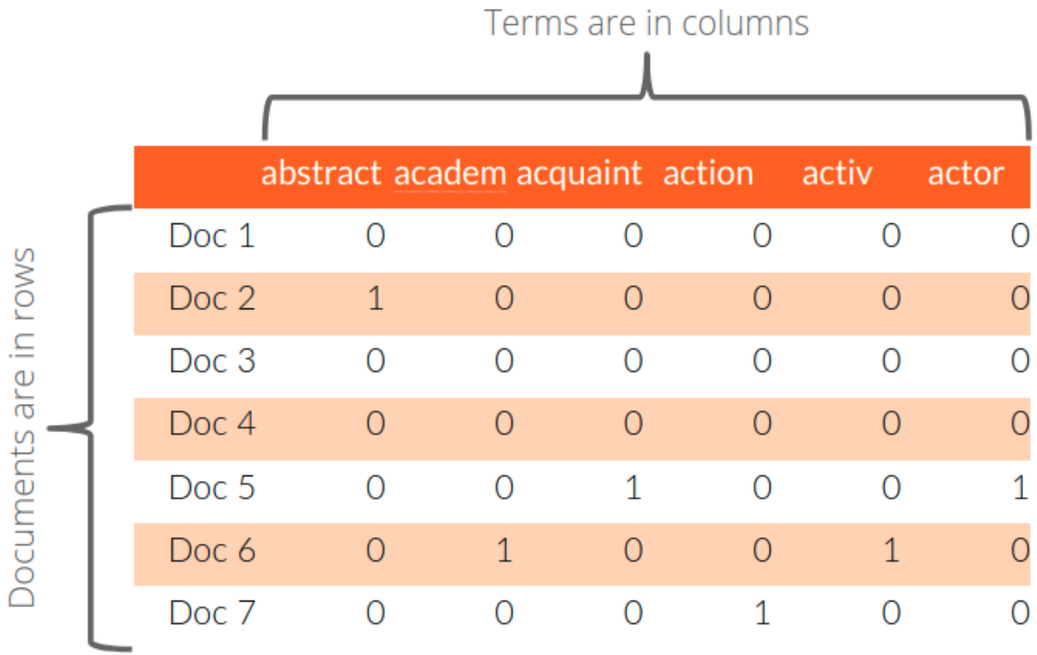
Field trip

- View ***https://databasic.io/en/wordcounter/***
- Select your favorite artist under the “use a sample” menu and hit “COUNT”
- Does the data tell a story about the selected artist?
- What additional type of analysis of this data might be interesting?

Module completion checklist

Objective	Complete
Create Term-Document Matrix	
Explore the distribution of words in corpus	

What is a Document-Term Matrix (DTM)?

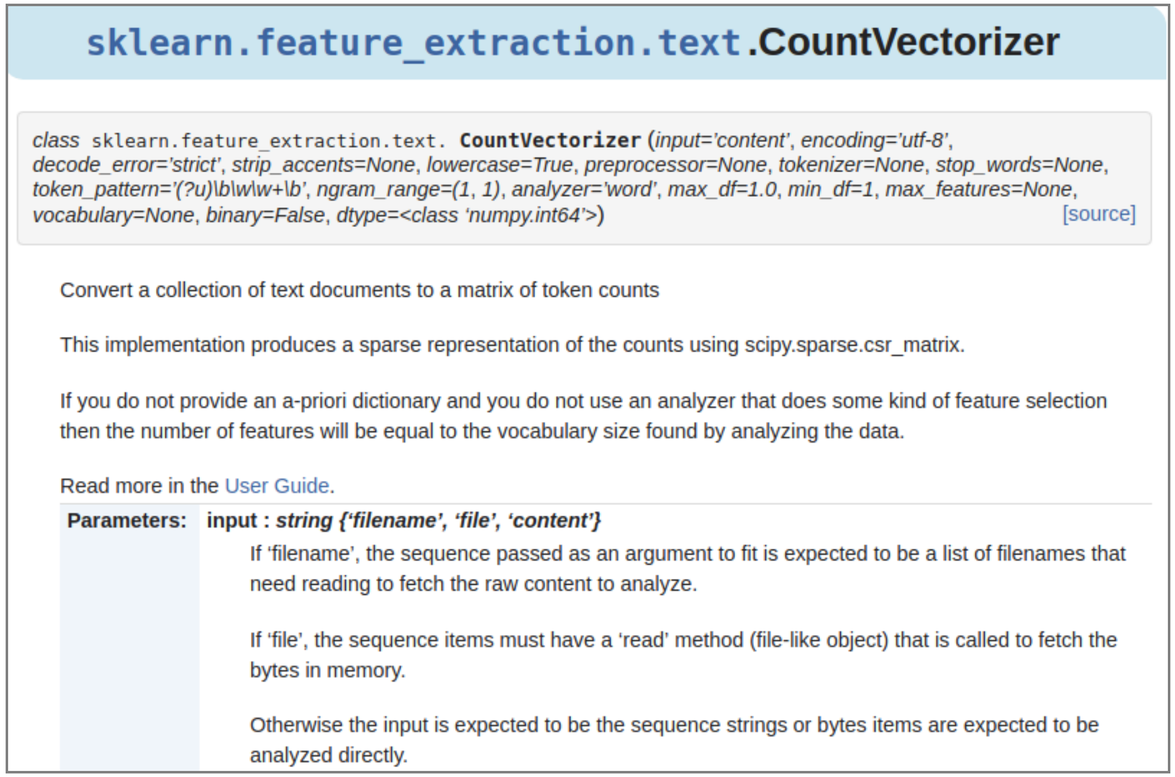


	abstract	academ	acquaint	action	activ	actor
Doc 1	0	0	0	0	0	0
Doc 2	1	0	0	0	0	0
Doc 3	0	0	0	0	0	0
Doc 4	0	0	0	0	0	0
Doc 5	0	0	1	0	0	1
Doc 6	0	1	0	0	1	0
Doc 7	0	0	0	1	0	0

- **Document-term matrix** is simply a matrix of unique words counted in each document:
  - Documents are arranged in rows
  - Unique terms are arranged in columns
- The corpus **vocabulary** consists of all of the unique terms (i.e. column names of DTM) and their total counts across all documents (i.e. column sums)
- A Term-Document Matrix will be just the transpose of the Document-Term Matrix, with terms in rows and documents in columns

Create DTM with CountVectorizer

- Another very powerful platform in Python is `scikit-learn`, it is used heavily for machine learning. You can find complete documentation ***here***.
- To create a Document-Term Matrix, we will use `CountVectorizer` from `scikit-learn` library's `feature_extraction` module for working with text



sklearn.feature_extraction.text.CountVectorizer

class sklearn.feature_extraction.text. **CountVectorizer** (*input='content', encoding='utf-8', decode_error='strict', strip_accents=None, lowercase=True, preprocessor=None, tokenizer=None, stop_words=None, token_pattern='(?u)\b\w\w+\b', ngram_range=(1, 1), analyzer='word', max_df=1.0, min_df=1, max_features=None, vocabulary=None, binary=False, dtype=<class 'numpy.int64'>*) [source]

Convert a collection of text documents to a matrix of token counts

This implementation produces a sparse representation of the counts using scipy.sparse.csr_matrix.

If you do not provide an a-priori dictionary and you do not use an analyzer that does some kind of feature selection then the number of features will be equal to the vocabulary size found by analyzing the data.

Read more in the User Guide.

Parameters: **input : *string {'filename', 'file', 'content'}***

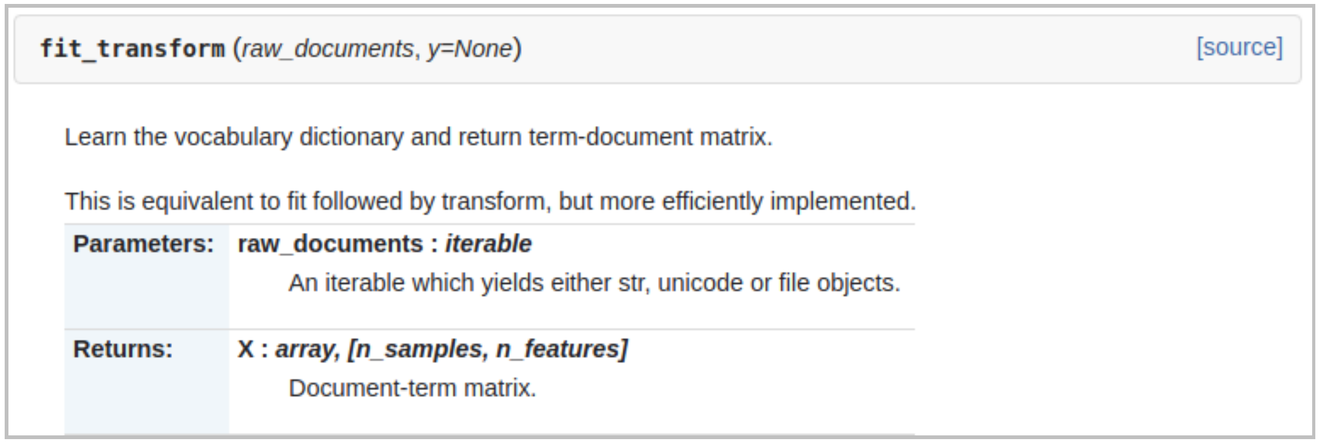
If 'filename', the sequence passed as an argument to fit is expected to be a list of filenames that need reading to fetch the raw content to analyze.

If 'file', the sequence items must have a 'read' method (file-like object) that is called to fetch the bytes in memory.

Otherwise the input is expected to be the sequence strings or bytes items are expected to be analyzed directly.

Create DTM with CountVectorizer (cont'd)

- It takes a `list` of character strings that represent the documents as the main argument, passed to its `fit_transform()` method:
  - `.fit_transform(list_of_documents)`
- It returns a 2D array (i.e. a matrix) with documents in rows and terms in columns - the **DTM**



`fit_transform` (*raw_documents*, *y=None*) [source]

Learn the vocabulary dictionary and return term-document matrix.

This is equivalent to fit followed by transform, but more efficiently implemented.

Parameters:	**raw_documents : *iterable*** An iterable which yields either str, unicode or file objects.
Returns:	**X : *array, [n_samples, n_features]*** Document-term matrix.

- For more information on this module, see ***scikit-learn's official documentation on this module***

Create a DTM

```
# Initialize `CountVectorizer`.
vec = CountVectorizer()
```

```
# Transform the list of documents clean documents `df_clean_list` into DTM.
X = vec.fit_transform(df_clean_list)
print(X.toarray()) #<- to show output as a matrix
```

```
[[0 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]]
```

- To get a list of names of columns (i.e. the unique terms in our corpus), we can use a utility method `.get_feature_names_out()`

```
print(vec.get_feature_names_out()[:10])
```

```
['abduct' 'abl' 'abo' 'absente' 'abus' 'academ' 'accept' 'access'
 'accessori' 'accommod']
```

Create a DTM (cont'd)

- Let's convert the matrix into a dataframe, where rows are IDs of the documents and columns are unique words that appear in those documents

```
# Convert the matrix into a Pandas DataFrame for easier manipulation.
DTM = pd.DataFrame(X.toarray(), columns = vec.get_feature_names_out())
print(DTM.head())
```

```
   abduct  abl  abo  absente  abus  academ  accept  access  ...  year  yell  yet  york
young  yuan  zimbabw  zykera
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
2       0    0    0        0     0       0       0       0  ...     0     0    0     0
1     0        0       0
3       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

[5 rows x 1921 columns]
```

DTM to dictionary of total word counts

- NLTK word frequency visualization functions work with dictionaries
- **Before we convert our DTM to a dictionary**, let's create a convenience function that sorts all words in descending order by counts and displays the first n entries
- We use `lambda` within the function

```
# Create a convenience function that sorts and looks at first n-entries in the dictionary.
def HeadDict(dict_x, n):
    # Get items from the dictionary and sort them by
    # value key in descending (i.e. reverse) order
    sorted_x = sorted(dict_x.items(),
                      reverse = True,
                      key = lambda kv: kv[1])

    # Convert sorted dictionary to a list.
    dict_x_list = list(sorted_x)

    # Return the first `n` values from the dictionary only.
    return(dict(dict_x_list[:n]))
```

DTM to dictionary of total word counts (cont'd)

```
# Sum frequencies of each word in all documents.
DTM.sum(axis = 0).head()
```

```
abduct     1
abl        1
abo        1
absente    1
abus       2
dtype: int64
```

```
# Save series as a dictionary.
corpus_freq_dist = DTM.sum(axis = 0).to_dict()
```

```
# Glance at the frequencies.
print(HeadDict(corpus_freq_dist, 6))
```

```
{'said': 38, 'new': 36, 'presid': 28, 'year': 27, 'friday': 22, 'govern': 22}
```

“Bag-of-words” analysis: key elements

- We have one more step remaining to learn, we will cover this next!

What we need	What we have learned
A corpus of documents cleaned and processed in a certain way • All words a converted to lowercase • All punctuation, numbers and special characters are removed • Stopwords are removed • Words are stemmed to their root form	✔
A Document-Term Matrix (DTM): with counts of each word recorded for each document	✔
A transformed representation of a Document-Term Matrix (i.e. weighted with TF-IDF weights)	

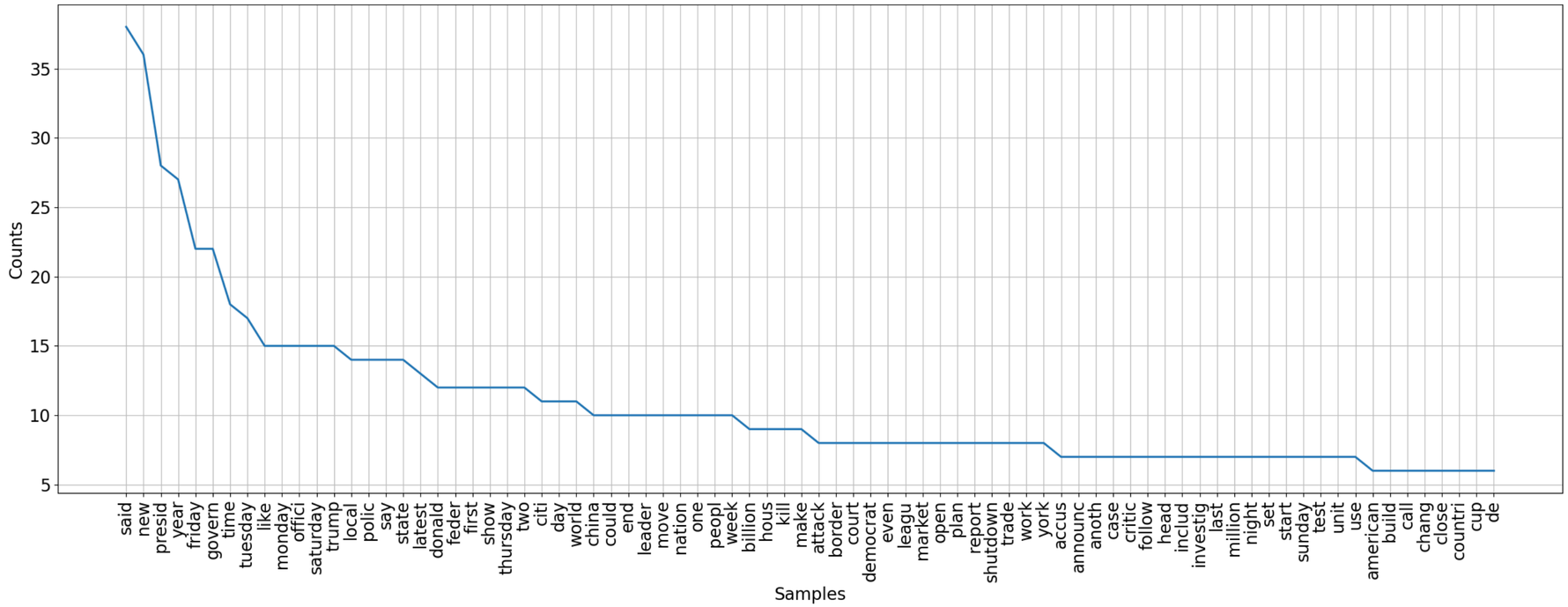
Module completion checklist

Objective	Complete
Create Term-Document Matrix	✔
Explore the distribution of words in corpus	

Plot distribution of words in document corpus

```
# Save as a FreqDist object native to nltk.
corpus_freq_dist = nltk.FreqDist(corpus_freq_dist)
```

```
# Plot distribution for the entire corpus.
plt.figure(figsize = (50, 10))
corpus_freq_dist.plot(80)
```



Visualizing words using n-grams

- An n-gram is a sequence of words that occur together in a sentence
- This concept is used extensively in the field of Natural Language Processing to build n-gram based models which have various use cases like:
  - Predicting next words in a sentence based on their probability
  - Correcting spelling based errors in a sentence
- We're not going to dig deeper into these models in this course, but let's learn how to create n-grams using `nltk` package!

Visualizing words using n-grams

- An n-gram is called:
  - **uni-gram** when the value of n is set to 1,
  - **bi-gram** when n = 2,
  - **tri-gram** when n = 3, and so on
- To create an n-gram, we will use `ngrams` from `nltk' library'sutil` module. You can find the complete documentation for the ngrams ***here***

`nltk.util.`**`ngrams`**(*sequence*, *n*, *pad_left=False*, *pad_right=False*, *left_pad_symbol=None*, *right_pad_symbol=None*) [source]

Return the ngrams generated from a sequence of items, as an iterator. For example:

```
>>> from nltk.util import ngrams
>>> list(ngrams([1,2,3,4,5], 3))
[(1, 2, 3), (2, 3, 4), (3, 4, 5)]
```

Wrap with list for a list version of this function. Set pad_left or pad_right to true in order to get additional ngrams:

```
>>> list(ngrams([1,2,3,4,5], 2, pad_right=True))
[(1, 2), (2, 3), (3, 4), (4, 5), (5, None)]
>>> list(ngrams([1,2,3,4,5], 2, pad_right=True, right_pad_symbol='</s>'))
[(1, 2), (2, 3), (3, 4), (4, 5), (5, '</s>')]
>>> list(ngrams([1,2,3,4,5], 2, pad_left=True, left_pad_symbol='<s>'))
[('<s>', 1), (1, 2), (2, 3), (3, 4), (4, 5)]
>>> list(ngrams([1,2,3,4,5], 2, pad_left=True, pad_right=True, left_pad_symbol='<s>', right_pad_symbol='</s>'))
[('<s>', 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, '</s>')]
```

Parameters

- **sequence** (*sequence or iter*) – the source data to be converted into ngrams
- **n** (*int*) – the degree of the ngrams
- **pad_left** (*bool*) – whether the ngrams should be left-padded
- **pad_right** (*bool*) – whether the ngrams should be right-padded
- **left_pad_symbol** (*any*) – the symbol to use for left padding (default is None)
- **right_pad_symbol** (*any*) – the symbol to use for right padding (default is None)

Return type

sequence or iter

Visualizing words using n-grams

- Let's create a basic bi-gram and a tri-gram using the `ngrams` function

```
from nltk.util import ngrams

print(df_clean_list[0])
```

```
nick kyrgio start brisban open titl defens battl victori american ryan harrison open round
tuesday
```

```
word = df_clean_list[0].split()
print(list(ngrams(word, 2)))  #<- set value of n as 2
```

```
[('nick', 'kyrgio'), ('kyrgio', 'start'), ('start', 'brisban'), ('brisban', 'open'),
('open', 'titl'), ('titl', 'defens'), ('defens', 'battl'), ('battl', 'victori'), ('victori',
'american'), ('american', 'ryan'), ('ryan', 'harrison'), ('harrison', 'open'), ('open',
'round'), ('round', 'tuesday')]
```

```
print(list(ngrams(word, 3)))  #<- set value of n as 3
```

```
[('nick', 'kyrgio', 'start'), ('kyrgio', 'start', 'brisban'), ('start', 'brisban', 'open'),
('brisban', 'open', 'titl'), ('open', 'titl', 'defens'), ('titl', 'defens', 'battl'),
('defens', 'battl', 'victori'), ('battl', 'victori', 'american'), ('victori', 'american',
'ryan'), ('american', 'ryan', 'harrison'), ('ryan', 'harrison', 'open'), ('harrison',
'open', 'round'), ('open', 'round', 'tuesday')]
```

Convenience function to generate n-grams

- Let's create a convenience function to generate bi-grams, tri-grams and four-grams for a subset of documents in our corpus!

```
def generate_ngrams(df_clean_list):
  for i in range(len(df_clean_list)):
      for n in range(2, 4):
          n_grams = ngrams(df_clean_list[i].split(), n)
          for grams in n_grams:
              print(grams)
```

```
generate_ngrams(df_clean_list[0:10])
```

```
('nick', 'kyrgio')
('kyrgio', 'start')
('start', 'brisban')
('brisban', 'open')
('open', 'titl')
('titl', 'defens')
('defens', 'battl')
('battl', 'victori')
('victori', 'american')
('american', 'ryan')
('ryan', 'harrison')
('harrison', 'open')
('open', 'round')
('round', 'tuesday')
('nick', 'kyrgio', 'start')
('kyrgio', 'start', 'brisban')
('start', 'brisban', 'open')
('brisban', 'open', 'titl')
```

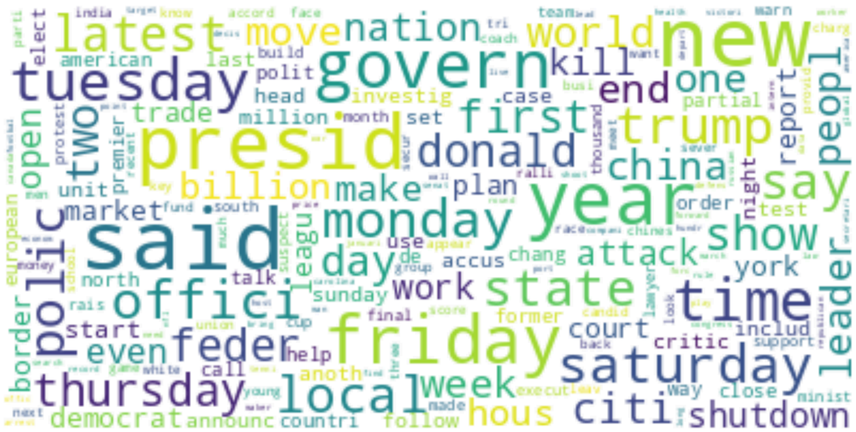
Visualizing word counts with word clouds

- World cloud is an effective way of visualizing word counts

```
# Construct a word cloud from corpus.
wordcloud = WordCloud(max_font_size = 40,
background_color = "white", collocations = False)
wordcloud = wordcloud.generate('
'.join(df_clean_list))
```

```
plt.figure(figsize = (27, 20)) # Plot the cloud
using matplotlib.
plt.imshow(wordcloud, interpolation = "bilinear")
plt.axis("off")
```

- What words are most common in the text data that we just cleaned?
- Based on that, what do you think these documents focus on?



Knowledge check

Exercise

You are now ready to try Tasks 7-9 in the Exercise for this topic

Module completion checklist

Objective	Complete
Create Term-Document Matrix	✔
Explore the distribution of words in corpus	✔

Text Processing: Topic summary

In this part of the course, we have covered:

- The need for text processing and the tools used to perform it
- Definition and implementation of text processing steps
- Word distribution in a corpus

Congratulations on completing this module!