

Text Encoding and Semantic Representation

Text analysis

marilena.daquino2@unibo.it | https://github.com/marilenadaquino/tesr_dhdk

Homework, all good?

Transform XML/TEI to HTML

Given the XML file, create a XSLT file to transform it in a valid HTML file.

Use the command-line program or the python script or the online tool to validate the transformation.

```
1 <xsl:stylesheet version="1.0"
2   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
3   xmlns:tei="http://www.tei-c.org/ns/1.0">
4   <xsl:template match="/">
5     <html>
6       <head>
7         <title><xsl:value-of select="tei:TEI/tei:text/tei:body/tei:div/tei:head/text()"/></title>
8       </head>
9       ...
10    </html>
11  </xsl:template>
12 </xsl:stylesheet>
```

Pay attention to the use of namespaces and prefixes!

Text analysis

What is it?

Text analysis addresses the **quantitative study** of text and words. It answers questions like frequency of words, length of sentence, and presence or absence of words.

It is particularly valuable in instances where there is a need to process large volumes of text-based data. Text analysis can enable us to consider, for example, hundreds of different features within a million books, **revealing patterns** inaccessible to a human reader.

Such approaches are often called also Text Analytics, Text mining, or Distant reading.

Text analysis

Tasks

Some of the most common tasks include the following:

Word frequency. Count the occurrences of words in a text.

N-grams. Collections of words that appear together.

Collocation. Sequences of words (n-grams) that co-occur together meaningfully “artificial intelligence”.

Concordance. List of occurrences of a word with the surrounding words.

Document classification. Create weighted lists of words that may represent topics.

Named Entity Recognition. Detect real-world entities in texts (e.g. people, places, dates).

Disclaimer

You are not required to perform the following tasks in your project

We are going to have an overview of tasks that characterise Text Analysis using Python. Some of the tasks described are also subject of another course, which is the one evaluating your competencies. For the sake of this exam you **may** perform one or more tasks (e.g. Named entity recognition) to enrich your final dataset. However, **it is not mandatory**.

Text analysis

Corpus preparation

Before calculating word frequency, we need to prepare our text corpus

- Open a terminal window and install NLTK `pip install nltk`
- Download from the repo of the course the file **6_led.txt**
- Create a new python file in the same folder

Text analysis

Read file

Open the file and access its content as a string.

```
1 # open the file with the text corpus
2 f = open("6_led.txt", "r")
3 text = f.read()
```


Text analysis

Transform to text

For the sake of the tutorial, we are going to use a plain text file. However, you will have to work on XML files as input documents, hence you will have to transform them into .txt.

See an example python-based transformation in the file **6_xml2txt.py**.

Text analysis

Tokenization

We need a version of our text where words (tokens) are separated from symbols, numbers, and punctuation marks. We download a set of punctuation marks and we use **word_tokenize()** on the original text to return a list of words.

Tokenization is a smarter way to split() our text into sequences of words. It does not remove punctuation or stop words though.

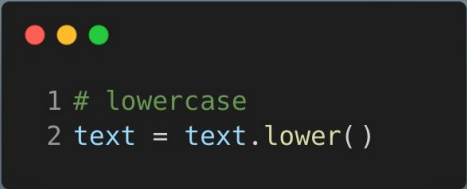
```
1 from nltk import word_tokenize
2
3 # open the file with the text corpus
4 f = open("6_led.txt", "r")
5 text = f.read()
6
7 # tokenization
8 words = word_tokenize(text)
9 print(words[:20])
```

```
1 ['The', 'profound', 'impression', 'made', 'upon', 'a',
  'crowded', 'congregation', 'in', 'St.', 'Paul', "'s",
  'Cathedral', 'has', 'already', 'been', 'mentioned', '.', 'An',
  'eloquent']
```

Text analysis

Lowercase

Sometimes is important to normalise the text and remove disturbing variables, e.g. the lower/uppercase of words. The function `lower()` transforms a string in lowercase.



```
1 # lowercase
2 text = text.lower()
```

Text analysis

Punctuation and stop words

To remove punctuation and stop words there are several ways. We use the module **string** to get a list of punctuation marks and the set of **stop words** provided by nltk to remove stop words. We use a list comprehension to get the final list of words.

```
1 import string
2 import nltk
3 from nltk import word_tokenize
4 from nltk.corpus import stopwords
5
6 # download the list of stop words (in english)
7 # only the 1st you run the code
8 nltk.download('stopwords')
9
10 # create sets of punctuation marks and stop words
11 stop_words = set(stopwords.words('english'))
12 excluded_punct = set(string.punctuation)
13
14 # clean tokens from punctuation and stop words
15 clean_text = [w for w in words if w not in excluded_punct and w not in stop_words]
16 print(clean_text[:20])
```

Text analysis

Normalization

All together our code to normalize the text looks like this. The object of our analysis is a list of words.

More operations could be performed, e.g. **lemmatization**, which extracts the root of words, e.g. “rocks” > “rock”, “went” > “be”

```
1 import string
2 import nltk
3 from nltk import word_tokenize
4 from nltk.corpus import stopwords
5
6 # download the list of stop words (in english)
7 # only the 1st you run the code
8 nltk.download('stopwords')
9
10 # create sets of punctuation marks and stop words
11 stop_words = set(stopwords.words('english'))
12 excluded_punct = set(string.punctuation)
13
14 # open the file with the text corpus
15 f = open("6_led.txt", "r")
16 text = f.read()
17
18 # lowercase
19 text = text.lower()
20
21 # tokenization
22 words = word_tokenize(text)
23 print(words[:20])
24
25 # clean tokens from punctuation and stop words
26 clean_text = [w for w in words if w not in excluded_punct and w not in stop_words]
27 print(clean_text[:20])
```

Text analysis

Word frequency

Counting occurrences of words in a text (corpus) can be relevant to a number of decision-making processes, e.g. a teacher wants to adopt a vocabulary that is likely to be understood by the most (word frequency effect), a journalist wants to improve readability of their articles using common sense terms, a content designer wants to reuse the vocabulary of customers, etc.

It does not require any statistical knowledge, although is not always sufficient to determine the most relevant keywords of a text.

Text analysis

Word frequency

Once the normalization has been performed, there are several methods to calculate word frequency:

- Use **Counter()** from the built-in library collections
- Use **FreqDist()** from the library nltk

```
1 from collections import Counter
2 from nltk import FreqDist
3
4 # word frequency with Counter
5 freq = Counter(clean_text).most_common(20)
6 print(freq)
7
8 # word frequency with FreqDist
9 fdist1 = FreqDist(clean_text).most_common(20)
10 print(fdist1)
```

Text analysis

Word frequency

Results are somehow meaningful.

The corpus is about military context, bands' concerts, and actions of their listeners.

```
1 [('military', 31), ('band', 27), ('music', 19), ('would', 11), ('played', 11),  
  ('first', 11), ('playing', 10), ('went', 10), ('long', 9), ('...', 9),  
  ('great', 8), ('one', 8), ('two', 8), ('heard', 7), ('came', 7), ('drums', 7),  
  ('bands', 7), ('see', 7), (''s', 6), ('``', 6)]
```


Text analysis

N-grams

N-grams are sequences of adjacent words/letters/syllables appearing together in a text.

$N = 1$ - unigrams ; $N = 2$ - bigrams ; $N = 3$ - trigrams

N-grams are used in NLP tasks where the sequence of words is relevant (e.g. sentiment analysis, text generation). However, reusing them can be challenging, because of the **data diversity**, i.e. the sparsity of ngrams in a text corpus leads to low probabilities to predict those ngrams in a new context (e.g. autocompletion).

Text analysis

N-grams

Like with word frequency, there are several ways to calculate ngrams.

In NLTK we can use the **ngrams()** function to extract n-grams. We can also calculate their frequency to understand most common ngrams (in the example, we calculate bigrams).

```
1 from nltk import ngrams
2
3 # get bigrams and calculate frequency of bigrams
4 n_grams = ngrams(clean_text, 2)
5 fdist3 = FreqDist(n_grams).most_common(20)
6 print(fdist3)
```

Text analysis

N-grams

Looking at the results, we appreciate that some bigrams are meaningful, and help to frame the topic (military bands' concerts, probably in UK "god save the queen"). Other bigrams could be pruned (e.g. by frequency).

```
1 [ (('military', 'band'), 14), (('military', 'music'), 3), (('body', 'church'), 3),  
2 [ (('band', 'playing'), 3), (('music', 'band'), 2), (('much', 'one'), 2), (('play', 'marches'), 2),  
   [ (('would', 'play'), 2), (('bass', 'drums'), 2), (('captain', 'waterhouse'), 2), (('bass', 'oboe'), 2),  
   [ (('knew', 'first'), 2), (('god', 'save'), 2), (('save', 'queen'), 2), (('first', 'thing'), 2),  
   [ (('sang', 'song'), 2), (('ought', 'join'), 2), (('westminster', 'hall'), 2), (('arthur', 'bigge'), 2),  
   [ (('military', 'bands'), 2)]
```

Text analysis

Collocation

Some ngrams are not really **informative** (E.g. “and then”) and require measures to discard them and to detect the meaningful ones.

Collocation measures help to understand the **strength** of ngrams, to identify combinations of words that have a specific meaning in a given **context**, and to identify **keywords** that are formed by multiple words. Most measures are based on the probability of ngrams.

Popular association measures include: **PMI, student's t-test, chi-squared test, and likelihood-ratio test.**

Text analysis

Collocation

Chi-squared

Measures the significance of bigrams.
Based on the frequencies of the observed and expected occurrences of the two words (assuming they are independent).

Unreliable in small corpora.

PMI

(Pointwise Mutual Information) takes into account words co-occurrence in a corpus with respect to their individual frequencies.

Text analysis

Collocation

We use the collection of collocation measures provided by the class **BigramAssocMeasures()** of NLTK and we use another class to find bigrams (**BigramCollocationFinder**).

The new object comes with a method call **score_ngrams**, which accepts as parameter a collocation measure.

```
1 from nltk import BigramAssocMeasures, BigramCollocationFinder
2
3 # initialize collocation measures for bigrams
4 bigrams = nltk.collocations.BigramAssocMeasures()
5
6 # find bigrams in the corpus
7 finder = BigramCollocationFinder.from_words(clean_text)
8
9 # score bigrams w/ Chi-squared
10 print("chi-squared")
11 scored_chi = finder.score_ngrams(bigrams.chi_sq)
12 for bigram, score in scored_chi[:5]:
13     print(bigram, score)
14
15 # PMI score
16 print("PMI")
17 scored_pmi = finder.score_ngrams(bigrams.pmi)
18 for bigram, score in scored_pmi[:5]:
19     print(bigram, score)
```

Text analysis

Collocation

Results are not so informative as we'd expect... While these words are highly collocated, the expressions are also very infrequent.

We can apply filters, e.g. ignoring all bigrams which occur less than 3 times.

Now results look more interesting.

```
1 chi-squared
2 ('1845', 'moment') 2079.0
3 ('1910.', 'dear') 2079.0
4 ('1st', 'september') 2079.0
5 ('20', 'sept.') 2079.0
6 ('absent', 'bandsmen') 2079.0
7 PMI
8 ('1845', 'moment') 11.02167404285837
9 ('1910.', 'dear') 11.02167404285837
10 ('1st', 'september') 11.02167404285837
11 ('20', 'sept.') 11.02167404285837
12 ('absent', 'bandsmen') 11.02167404285837
```

```
1 finder.apply_freq_filter(3)
2 print(finder.nbest(bigrams.chi_sq, 10))
3 print(finder.nbest(bigrams.pmi, 10))
```

```
1 [('body', 'church'), ('military', 'band'),
   ('band', 'playing'), ('military', 'music')]
```

Text analysis

Concordance

Concordance is the listing of each occurrence of a word (or pattern) in a text or corpus, presented with the words surrounding it.

The use of concordances is essentially a manual task for human analysts. Since the occurrences can be many, the list can be cut to a relevant number of exemplars.

Typically concordance is leveraged in language learning environments.

Text analysis


Concordance

To print the concordance list we reuse the class `Text()` from NLTK.

The function **`concordance()`** accepts 3 parameters: the term to search, the number of characters per line, and the number of lines to show.

```
1 from nltk.text import Text
2
3 # concordance
4 tokens_to_be_analysed = Text(clean_text)
5 tokens_to_be_analysed.concordance("military", 40, 20)
```

words



```
1 aps , naval and military uniforms and ac
2 assadable , and military exercises alway
3 , some russian military singers . they
4 arched with the military band before us
5 of the austrian military band , which pl
6 he day -- a big military band -- that wo
7 ganisation with military music . but the
8 llish , faintly military , not young , a
9 apparition of a military band ... i sa
10 like any other military band . they pla
11 over . when the military band performed
12 life . and the military reviews with th
13 is newly-formed military band which , at
14 them . it wasnt military music but enoug
15 e and free from military busyness ! i ha
16 ir and having a military band to supply
17 as crowded with military men and beautif
18 . then came the military bands , with th
19 chair , and the military band blasted ou
20 d themselves in military array , and ' g
```

Notice that the concordance should be performed on the original text rather than the normalised one

Text analysis

Document classification

When examining corpora, documents can be classified according to their content. TF-IDF and topic modelling are some of the most used techniques to classify texts and to compute similarity.

TF-IDF assigns high weight to terms which have (1) a high term frequency in a given document) and (2) a low document frequency in the whole collection of documents.

In other terms, words that occur in high frequency in a specific document but rarely in the rest of the corpus achieve high TF-IDF scores, while words that occur in lower frequency in a specific document but commonly in the rest of the corpus achieve low TF-IDF scores.

So doing, TF-IDF highlights topics that characterise a document among others.

Text analysis

TF-IDF

Term frequency (TF) is how often a word appears in a document, divided by how many words there are.

$TF(t) = (\text{Number of times term } t \text{ appears in a document}) / (\text{Total number of terms in the document})$

Inverse document frequency (IDF) is how unique or rare a word is.

$IDF(t) = \log_e(\text{Total number of documents} / \text{Number of documents with term } t \text{ in it})$

Find examples and applications [here](#) ; [here](#)

Text analysis

TF-IDF

There are several implementations of TF-IDF in Python libraries, we are going to use the one provided by sklearn. We also install numpy, that we will use to access results

In the terminal, install sklearn and numpy library

```
pip install sklearn
```

```
pip install numpy
```

Text analysis

TF-IDF

The class `TfidfVectorizer` takes in input a list of parameters:

```
1 from sklearn.feature_extraction.text import TfidfVectorizer
2
3 # split the corpus
4 texts = [t for t in text.split('\n') if t and len(t) > 1]
5
6 # Initialize a TFIDF object, applying some settings
7 tfidf = TfidfVectorizer(analyzer='word', sublinear_tf=True, max_features=500, tokenizer=word_tokenize)
8 tfidf = tfidf.fit(texts)
9
10 # Get the terms with the highest IDF score
11 inds = np.argsort(tfidf.idf_)[-10:]
12 top_IDF_tokens = [list(tfidf.vocabulary_)[ind] for ind in inds]
13
14 print(top_IDF_tokens)
```

- **analyzer**: can be words, characters, chars ngrams..
- **sublinear_tf**: It seems unlikely that 20 occurrences of a term in a document truly carry 20 times the same significance. Sublinear_tf is a variant of **tf**, that takes in the logarithm of the tf
- **max_features**: a vocabulary that only consider the top max_features ordered by term frequency across the corpus.
- **tokenizer**: we specify the tokenizer of NLTK

Text analysis

TF-IDF

We use the method **fit()** to a list of texts (we had to split our corpus first).

The returned object (called **tfidf** here) carries an array (a vector) that includes all the idf values. The array is returned by **tfidf.idf_** attribute.

```
1 from sklearn.feature_extraction.text import TfidfVectorizer
2
3 # split the corpus
4 texts = [t for t in text.split('\n') if t and len(t) > 1]
5
6 # Initialize a TFIDF object, applying some settings
7 tfidf = TfidfVectorizer(analyzer='word', sublinear_tf=True, max_features=500, tokenizer=word_tokenize)
8 tfidf = tfidf.fit(texts)
9
10 # Get the terms with the highest IDF score
11 inds = np.argsort(tfidf.idf_)[-1:][:10]
12 top_IDF_tokens = [list(tfidf.vocabulary_)[ind] for ind in inds]
13
14 print(top_IDF_tokens)
```

You can tune the n . of features to return different results

```
1 [2.09861229 2.38629436 2.09861229 1.69314718 1.87546874 1.87546874
2 1.87546874 1.08701138 1.87546874 2.79175947 1.
3 2.38629436 2.79175947 2.79175947 2.38629436 2.09861229 1.5389965
4 1.87546874 1.87546874 1.69314718 1.18232156 1.69314718 2.79175947
5 2.79175947 1.5389965 2.09861229 2.38629436 2.38629436 1.28768207
6 2.38629436 2.38629436 2.09861229 2.09861229 2.09861229 1.5389965
7 1.08701138 2.79175947 2.09861229 2.38629436 1.69314718 2.79175947
8 2.38629436 2.09861229 1.28768207 2.79175947 1.28768207 2.38629436
9 2.38629436 2.38629436 2.38629436 2.38629436 2.79175947 2.38629436
10 2.09861229 1.28768207 2.09861229 2.79175947 2.38629436 1.5389965
11 ... ]
```

Text analysis

TF-IDF

To access the array we use the method **argsort()** from the numpy library, which returns an **array of indices** of the same shape as the array that you would need to sort the input array in ascending order.

[::-1] reverses the list (so that we get scores in descending order)

[:10] extracts the first ten indexes of the list, i.e. the top 10 scores

```
1 arr = [2, 0, 1, 5, 4, 1, 9]
2 print(np.argsort(arr))
3 # returns
4 # [1 2 5 0 4 3 6]
5 # where 1 = 0 ; 2 = 1; 5 = 1 ; 0 = 2 ; 4 = 4 ; 3 = 5 ; and 6 = 9
```

```
1 inds = np.argsort(tfidf.idf_)[::-1][:10]
2 top_IDF_tokens = [list(tfidf.vocabulary_)[ind] for ind in inds]
```

```
1 [160 419 386 149 309 187 155 379 156 157]
```

Text analysis

TF-IDF

The attribute **.vocabulary_** returns a dictionary of terms and their indices. We transform the dictionary in a list with only the keys and we extract the elements that appear at the indexes previously identified.

The returned terms are those characterising the documents in the collection.

```
1 {'the': 435, 'made': 238, 'a': 21, 'crowded': 111, 'in': 211,
  'st.': 418, 'paul': 317, 's': 4, 'cathedral': 87, 'has': 187,
  'been': 63, '.': 10, 'an': 35, 'by': 79, 'of': 293, 'london':
  235, '(': 5, 'to': 453, ')': 6, 'was': 473, 'occasion': 291,
  'and': 36, 'choir': 95, 'did': 116, 'music': 260, ',': 7,
  'selection': 391, 'direction': 11 ...}
```

```
1 inds = np.argsort(tfidf.idf_)[::-1][:10]
2 top_IDF_tokens = [list(tfidf.vocabulary_)[ind] for ind in inds]
```

```
1 ['drums', 'orchestra', 'greater', 'big', 'drive', 'very',
  'snare', 'bend', 'drum', 'mostly']
```


Text analysis

Named Entity Recognition

NER algorithms locate and classify named entities in text into predefined categories such as the names of persons, organizations, locations, expressions of times, quantities, monetary values, percentages, etc.

Most libraries rely on pre-trained models to classify entities that have never been categorised before. For instance, [en_web_core_sm](#) is a small English pipeline trained on written web text (blogs, news, comments), that includes vocabulary, syntax and entities.

Text analysis

NER

We first install Spacy

```
pip install spacy
```

And download the model for English

```
python -m spacy download en_core_web_sm
```

We load the model and apply the pipeline for text analysis (i.e. the method **nlp()**)

```
1 import spacy
2
3 f = open("6_led.txt", "r")
4 orig_text = f.read()
5
6 # download the Spacy model for English
7 nlp = spacy.load("en_core_web_sm")
8
9 doc = nlp(orig_text)
10 print([(X.text, X.label_) for X in doc.ents])
```

Text analysis

NER

We access its attribute **ents**, which is an object including extracted words (or chunks) and their category.

As you can see, there are some mistakes, but all in all, plenty of entities are recognised and correctly associated to a category.

```
1 import spacy
2
3 f = open("6_led.txt", "r")
4 orig_text = f.read()
5
6 # download the Spacy model for English
7 nlp = spacy.load("en_core_web_sm")
8
9 doc = nlp(orig_text)
10 print([(X.text, X.label_) for X in doc.ents])
```

```
1 [('"St. Paul's", 'ORG'),
2  ('Cathedral', 'ORG'),
3  ('Charles Macpherson', 'PERSON'),
4  ('Te Deum', 'PERSON'),
5  ('the Welsh Guards', 'LOC'),
6  ('Spaniards', 'NORP'),
7  ('Russian', 'NORP'),
8  ('six', 'CARDINAL'),
9  ('Schalmey', 'PERSON'),
10 ('Charles] Hewitts', 'PERSON'),
11 ('Charles] Grays', 'ORG'),
12 ('Austrian', 'NORP'),
13 ('three', 'CARDINAL'),
14 ('the week', 'DATE'),
15 ('the Piazza of St.Marc', 'ORG'),
16 ('the Odd Fellows', 'ORG'),
17 ('one', 'CARDINAL'),
18 ('the day', 'DATE'),
19 ('two', 'CARDINAL'),
20 ('first', 'ORDINAL'),
21 ('one', 'CARDINAL'),
22 ('New Orleans', 'GPE'),
23 ('two', 'CARDINAL'),
24 ('Beecham', 'PERSON'),
25 ('Mass of Life', 'WORK_OF_ART'),
26 ('Captain Waterhouse', 'PERSON') ...]
```

NLP

What is it?

It investigates the **semantics**, the linguistic use of text and the context of usage. It answers questions like the intention behind a sentence, people's linguistic habits, and classify texts according to one or more categories.

NLP

Tasks

Sentiment analysis. Classify texts according to their valence/polarity.

Text summarisation. Generate shorter versions of a text.

Text generation. Generate new text given an input

Speech recognition. Recognition and translation of spoken language into text

Natural Language Understanding. Understand input in the form of sentences using text or speech.

Machine translation. Automatically translate across languages.

NLP

Sentiment analysis

Sentiment analysis is a natural language processing technique that identifies the polarity of a given text. There are different flavors of sentiment analysis, but one of the most widely used techniques labels data into **positive, negative and neutral**.

To classify texts according to their sentiment, three approaches exist:

- Lexicon (or rule) based, e.g. Vader: predefined rules and heuristics to determine the sentiment
- Machine Learning: based on a set of labeled training data
- pre-trained neural network models: encode context and meaning of words witnessed in massive datasets of texts, and used to train an algorithm for the identification.

Small digression

What is a language model?

A language model is a probabilistic model of a natural language [\[Wikipedia\]](#). It generates the probability of combinations of words based on:

- Pure statistical models (probability-based)
- Vector-based models (including weighted words and meaning) - i.e. neural networks

Large Language Models (LLM) are neural networks based on **massive datasets of texts**, from which they learn the features characterising words and meanings. LLMs interpret this data by feeding it through an algorithm that establishes **rules for context** in natural language. Then, the model applies these rules in language tasks to accurately predict or produce new sentences.

NLP

Sentiment analysis

Install the library for sentiment analysis

```
pip install eng-spacysentiment
```

Use the new pipeline on short, sentence-like texts.

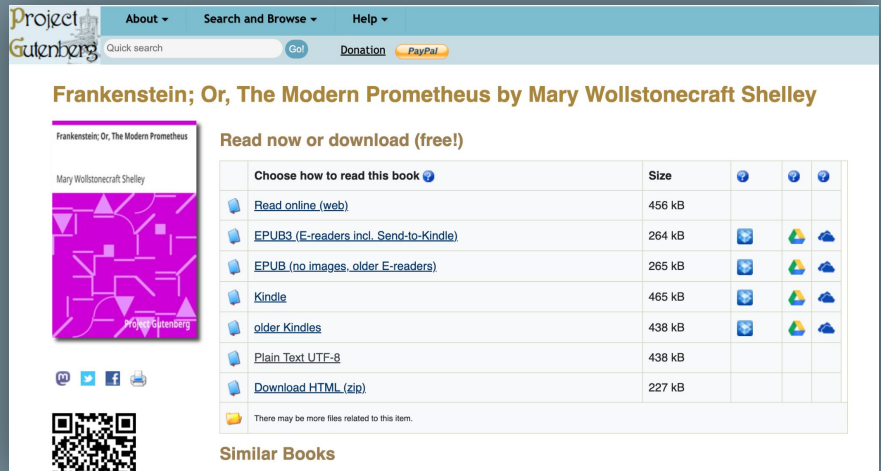
```
1 import eng_spacysentiment
2
3 nlp2 = eng_spacysentiment.load()
4 # apply it to the first text in the corpus
5 doc2 = nlp2(texts[0])
6 print(doc2.cats)
```


Exercise | Homework

Analyse your favourite book

1. Go to the [Gutenberg project](https://www.gutenberg.org/) website and pick your favourite book.
2. Access the web page of the chosen book and download the Plain text.
3. Create a new python file in the same folder.

Backup plan: download **6_frankenstein.txt**
from our repo



The screenshot shows the Project Gutenberg website interface. At the top, there's a navigation bar with 'About', 'Search and Browse', and 'Help' menus. Below this is a search bar and a 'Go!' button. The main heading is 'Frankenstein; Or, The Modern Prometheus by Mary Wollstonecraft Shelley'. To the left of the text is a book cover image. Below the cover are social media icons and a QR code. The main content area is titled 'Read now or download (free!)' and contains a table with download options.

Choose how to read this book	Size			
Read online (.web)	456 kB			
EPUB3 (E-readers incl. Send-to-Kindle)	264 kB			
EPUB (no images, older E-readers)	265 kB			
Kindle	465 kB			
older Kindles	438 kB			
Plain Text UTF-8	438 kB			
Download HTML (zip)	227 kB			

There may be more files related to this item.

Similar Books

Exercise | Homework

Analyse your favourite book

Write a python function to reads the .txt as a string and computes the following statistics

- The number of **tokens**
- The number of **unique tokens**
- The 20 most **frequent words** and their counting
- The 20 most **frequent bigrams** and their counting
- The 10 most significant **collocations** of bigrams
- The number of **sentences** (See [documentation](#))
- The number of **Chapters** or structures alike, if applicable (e.g. count the word “Chapter”). If no division exist, create an artificial one (e.g. split the text in chunks of characters)

Exercise | Homework

Analyse your favourite book

Topics

- Identify the 10 most significant themes with **TF-IDF**
- Print the **concordance** of the 10 terms (max. 20 occurrences per each term)

People

- Identify named entities (**NER**) and select only people (if no people are mentioned, use places)
- Manually group strings that correspond to the same person, search occurrences and return the **top 5 most mentioned people** (and their counting)

For each sentence in the book

- Perform **sentiment analysis** and return the **top 5 happiest/saddest sentences**