• Summary •	
Problem 1	**Basic corpus statistics**
Problem 2	**Keyword extraction – Prep. data**
Problem 3	**Keyword extraction**
Problem 4	**Document Categorization**

PROBLEMS

▶ Problem 1 – Basic corpus statistics

In this tutorial, we explore some of MATLAB's powerful capabilities for text processing tasks such as keyboard extraction and document classification. The tutorial draws heavily from the User's Guide for MATLAB's Text Analytics Toolbox™ and the second edition of Banchs's (2021) textbook *Text Mining with MATLAB*. Before we delve in any specifics, however, I provide a quick rundown of interesting insights into corpus linguistics that can be derived and illustrated with MATLAB's computational tools.

We begin by loading a simple English-language corpus to work with. I chose to use *The Complete Works of William Shakespeare*, a 1994 compilation that can be downloaded from Project Gutenberg or directly from one of our Google Drive folders. (Please bear in mind that the book is to be utilized for personal or academic use only; not all books available on Project Gutenberg are available in the public domain.) We can load the file text via MATLAB's *extractFileText* command:

```
fileName = "shaks12.pdf";
str = extractFileText(fileName);
```

Importantly, this PDF file contains junk text such as page numbers and legal reproduction disclaimers; while these pieces of text can be easily edited out for use in MATLAB, we will save time by leaving the PDF file unchanged. This has no effect on our initial analysis, which is corpus-based only. (We'll be much more careful when handling text data for keyword extraction and doc classification; see below.)

We proceed to lowercase the contents of the file:

```
str = lower(str);
```

Then, we replace all non-alphanumeric characters with whitespaces:

```
str = regexprep(str, '\W', ' ');
```

Then, we eliminate leading and trailing whitespaces:

```
str = strtrim(str);
```

Lastly, we are ready to segment the string *str* into a sequence of words/tokens:

```
words = split(str);
```

The number of words in the Shakespeare corpus is:

```
running_words = length(words)
running_words =

      930189
```

That is, the processed string amounts to approximately 930,000 words. This is not a large number, as some corpora used in modern text mining applications can easily contain dozens of millions of words. A second important computation pertains to the diversity of vocabulary words; to obtain the number of *unique* words in the *shaks.pdf* corpus, we employ the function *unique*:

```
vocabulary = unique(words);
vocabulary_words = length(vocabulary)
vocabulary_words =

       27224
```

The output, which is close to 27,200, is a mere 3 percent of the total number of words; this indicates that the corpus of running words is constituted of a relatively small number of words, and some words must occur several times over the course of the text. The frequency of repetition of a given word can be obtained by counting the number of times each word occurs in the whole data collection. Following Banchs (2021), we can take advantage of functions *unique* and *hist* to obtain the frequencies as follows:

```
[vocab, void, index] = unique(words);
freqs = hist(index, vocabulary_words);
whos index
whos freqs
Name            Size              Bytes   Class    Attributes
index         930189x1          7441512  double
Name            Size              Bytes   Class     Attributes
freqs          1x27224           217792   double
```

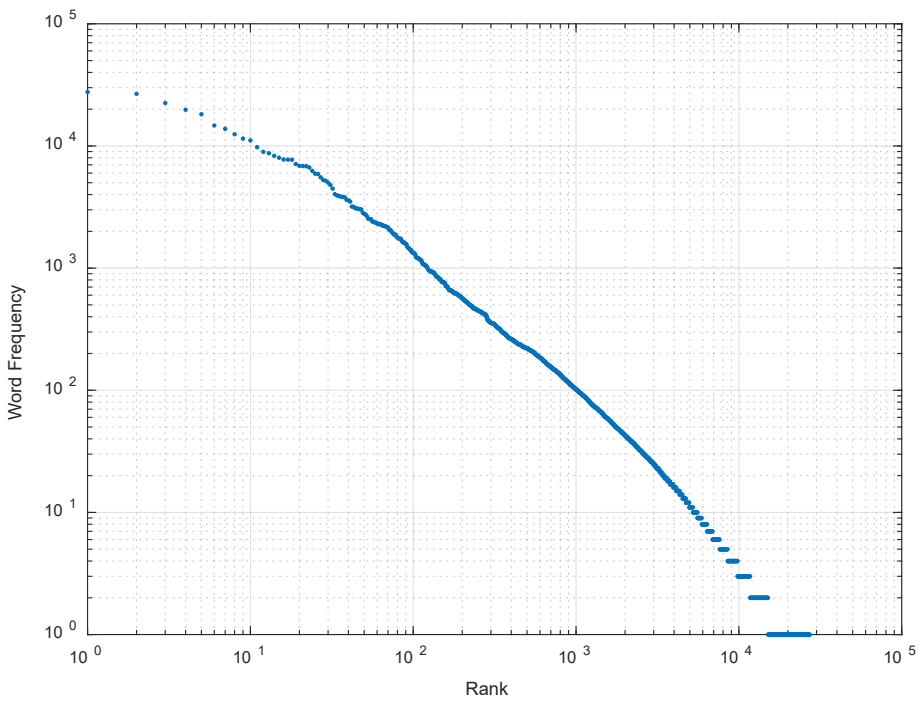
In the code snippet above, *unique* is used to extract the vocabulary. The *index* variable contains pointers to the corresponding word in *vocab*. Then, the function *hist* is used to construct a histogram with as many bins as the total amount of words in the vocabulary. The ensuing numeric array *freqs* contains the total number of times each word in the vocabulary occurs in the whole collection.

It remains to rank the frequency counts contained in *freqs*. To do so, we use the *sort* command:

```
[ranked_freqs, ranking_index] = sort(freqs, 'descend');
ranked_vocab = vocab(ranking_index);
```

We are then ready to construct the plot of word frequencies versus rank for the text data at hand:

```
set(figure, 'Color', [1,1,1], 'Name', 'Zipf''s Law');
loglog(ranked_freqs,'.'); grid on
xlabel('Rank'); ylabel('Word Frequency');
```



The resulting plot is shown above. Notice that a linear trend holds in this bilogarithmic word frequency versus rank plot, albeit with some deviation at the upper and lower extremes. Such a linear trend is known as *Zipf's law*[1] and appears to hold for natural discourse in any language.

Inspecting the upper tail of the Zipf law plot, it is clear that there are a few particularly important outlier words (points). To retrieve the, say, four uppermost outlier words, we type:

```
ranked_vocab(1:4)
ans =
  4×1 string array
    "the"
    "and"
    "i"
    "to"
```

Thus, the four most frequent words in Shakespeare's works seem to be *the*, *and*, *I*, and *to*, in decreasing order. The second, third and fourth most common words vary from document to document, but *the* is generally the most frequent word in most English communication. Let us add up the total number of occurrences of the four abovementioned most common words:

```
sum(ranked_freqs(1:4))
ans =
       96607
```

When combined, these four words alone occur nearly 100,000 times, which is about 10.4% of the entire Shakespeare corpus. The following table summarizes other cumulative statistics for increasingly larger subsets of the *n* most frequent words.

Rank interval	*n* = 1 to 4	*n* = 1 to 40	*n* = 1 to 400	*n* = 1 to 4000
Percentage of vocabulary	0.015%	0.15%	1.5%	15.0%
Total No. of occurrences	96,607	358,618	645,239	861,510
Percent of whole collection	10.39%	38.55%	69.37%	92.62%

Notice from the table that a mere 15% of the Shakespeare vocabulary accounts for over 92% of the entire corpus.

Another interesting observation, also due to Zipf, is that the most frequently used words in a language are often the shortest ones. Indeed, if we sample the words ranked, say, from 45 to 60 in the Shakespeare corpus and compare them with words ranked, say, 3000 to 3015, we obtain the following:

```
ranked_vocab(45:60)'

ans =

  1×16 string array

"lord"    "our"    "o"    "king"    "good"    "now"    "sir"
"from"    "they"    "at"    "come"    "ll"    "she"    "let"
"enter"    "here"

ranked_vocab(3000:3015)'

ans =

  1×16 string array

  Columns 1 through 12

"plebeians" "proceeding"  "rebel"    "rejoice"  "repose"
"rightly"    "roaring"    "safely"    "sandys"    "setting"    "sex"
"sheriff"

  Columns 13 through 16

 "signify"    "siward"    "stabs"    "stopp"
```

[1] In addition to linguistics, Zipf's law also occurs in other areas of knowledge. For instance, the ranked distribution of city populations within a country is also expected to follow a bilogarithmic pattern.

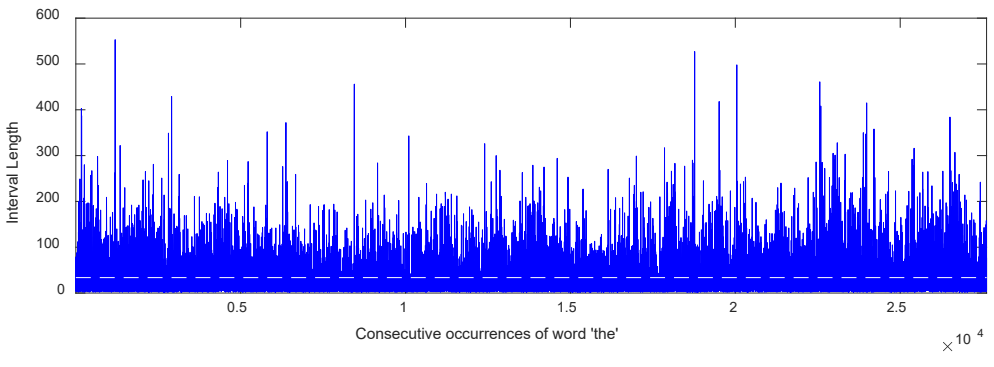
Clearly, the average length of the words of rank 45 to 60, which includes *sir*, *at*, and *she*, is generally less than the length of words ranking 3000 to 3015, which includes *proceeding*, *setting*, and *sheriff*. We can obtain further evidence of this by computing the average length of words at different points in the rank scale as follows:

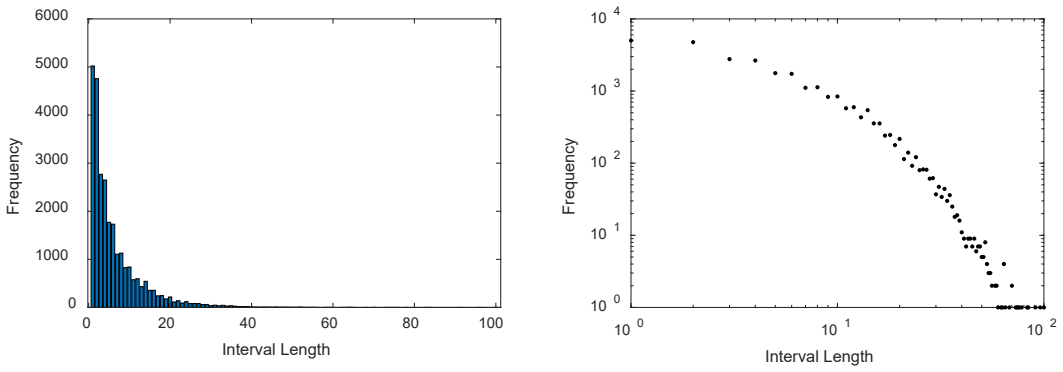
```
mean(strlength(ranked_vocab(45:90)))

ans =

    3.5652

mean(strlength(ranked_vocab(450:495)))

ans =

    6.0217

>> mean(strlength(ranked_vocab(3000:3045)))

ans =

    6.3478
```

As can be seen, the farther from 1 the rank range entered into MATLAB is, the greater the average length of the corresponding words. This reflects an "auto-compressing property" of language, whereby human communication has evolved to become more efficient by relying primarily on short words.

A second interesting property of language that can be illustrated in a MATLAB framework is the concept of *burstiness*. As noted by Banchs (2021), this property has to do with the fact that words within a text sequence have the tendency to repeat themselves following some specific patterns that resemble *bursts* of occurrences. This means that when we first encounter a new word in a given text sequence there is a high chance that we will find it again relatively close in the following segments of text. To investigate this, let us compute and plot the interval lengths between consecutive occurrences of the word *the* and make a histogram of such intervals. To do so, we will be using MATLAB's *diff* and *hist* functions for computing the interval lengths and their histogram, respectively.

```
% Gets the indexes for the occurrences of the word 'the'
locations = find(strcmp(words,'the'));
% Computes the lengths of the repetition intervals
intervals = diff(locations);
% Computes a 100-bin histogram
histogram_the = hist(intervals, 100);
hf = figure(3); %Creates a new figure
set(hf, 'Color', [1,1,1], 'Name', 'Intermittency Property');
% Plots interval lengths between consecutive repetitions of 'the'
subplot(2,1,1);
nvals = length(intervals);
plot(1:nvals, intervals, '-b', [1,nvals], ones(1,2)*mean(intervals),
'--w');
limits = axis; axis([1, nvals, limits(3:4)]);
ylabel('Interval Length');
xlabel('Consecutive occurrences of word ''the'''');
% Plots the histogram of interval lengths
subplot(2,2,3);
bar(histogram_the);
xlabel('Interval Length');
ylabel('Frequency');
% Plots the distribution (histogram) in logarithmic space
subplot(2,2,4);
loglog(1:length(histogram_the), histogram_the, '.k');
xlabel('Interval Length');
ylabel('Frequency');
```

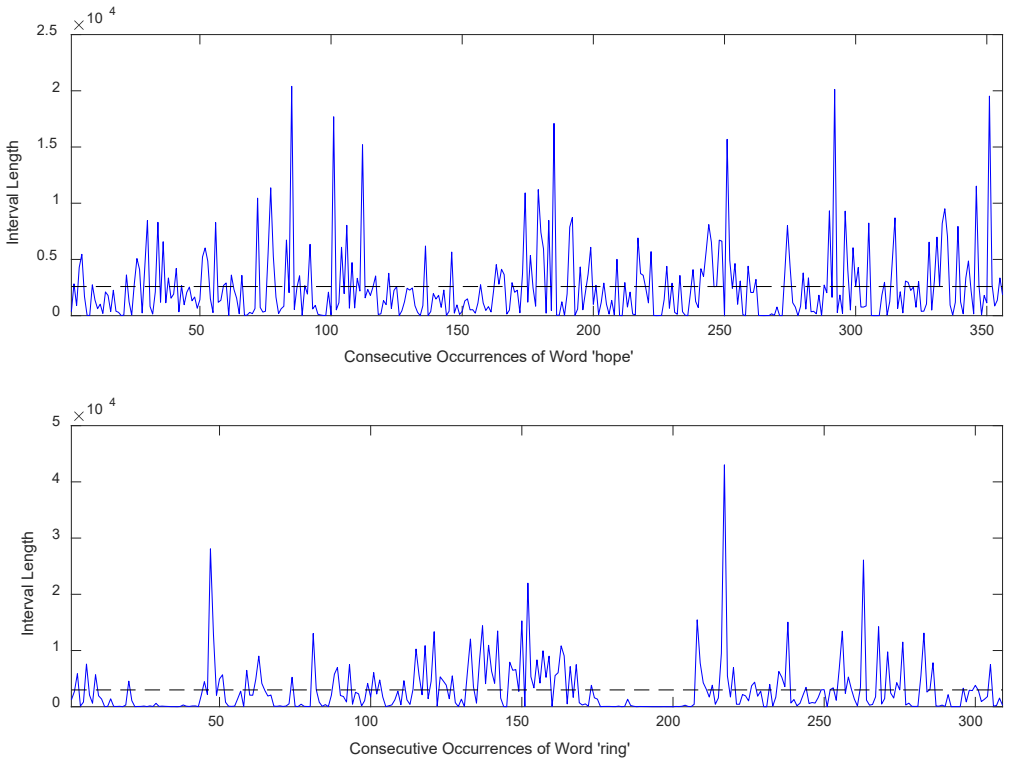




The figure shows the interval lengths between consecutive occurrences of the word *the* and their corresponding histogram in linear space (bottom left) and bilogarithmic space (bottom right). The upper plot illustrates our abovementioned notion that occurrences of *the* are not periodic, but rather 'bursty' and intermittent. The histogram on the lower left further illustrates that interval lengths between occurrences of the same word are not uniform. Finally, the bilogarithmic plot on the lower right shows that the ranked interval lengths for occurrences of the same word are approximately linearly distributed; this is yet another instance of Zipf's law in corpus linguistics.

We close this section by generating interval length plots akin to those obtained above, only this time considering two different words: *hope*, which is ranked at No. 300 in *vocab_words*, and *ring*, which is ranked at No. 345. Notice how a similarly intermittent behavior can be observed for both words.

```
% Writes words 'hope' and 'child' into a cell array
twoWords = {'hope','ring'};
% Retrieves their ranking positions
find(strcmp(ranked_vocab, twoWords{1}))
find(strcmp(ranked_vocab, twoWords{2}))
% Creates a new figure
set(hf, 'Color', [1,1,1], 'Name', 'Two additional interval length
plots');
% Generates the plots
for k = 1:2
    intervals = diff(find(strcmp(words, twoWords{k})));
    subplot(2,1,k);
    nvals = length(intervals);
    plot(1:nvals, intervals, '-b', [1, nvals],
ones(1,2)*mean(intervals), '--k');
    limits = axis;
    axis([1, nvals, limits(3:4)]);
    xlabel(sprintf('Consecutive Occurrences of Word ''%s'''',
twoWords{k}));
    ylabel('Interval Length')
end
```



▶ Problem 2 – Keyword extraction – Preparing data

We now turn to an elementary task in text mining: keyword extraction. We will be using three of the most influential novels of the nineteenth century: Melville's *Moby Dick*, Dostoyevsky's *Crime and Punishment*, and Oscar Wilde's *The Picture of Dorian Gray*. These three works can be downloaded for free from the Project Gutenberg website through the URLs listed below.

Book	URL for HTML file
Moby Dick	https://www.gutenberg.org/files/2701/2701-h/2701-h.htm
Crime and Punishment	https://www.gutenberg.org/files/2554/2554-h/2554-h.htm
The Picture of Dorian Gray	https://www.gutenberg.org/files/174/174-h/174-h.htm

The HTML files can be incorporated into MATLAB with the *webread* function:

```
url = "https://www.gutenberg.org/files/2701/2701-h/2701-h.htm";
code = webread(url);
```

The HTML code contains the relevant text inside <*p*> (paragraph) elements. To extract the relevant text, we may parse the HTML code using the *htmlTree* function and then finding all the elements with the element name "p."

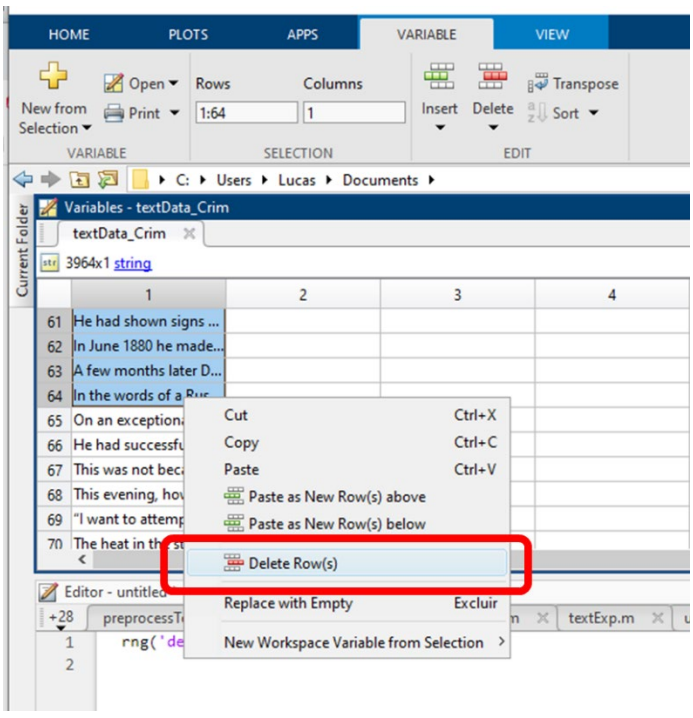
```
tree = htmlTree(code);
selector = "p";
subtrees = findElement(tree, selector);
```

Lastly, we extract the text data from the HTML subtrees using the *extractHTMLText* function and remove the empty elements.

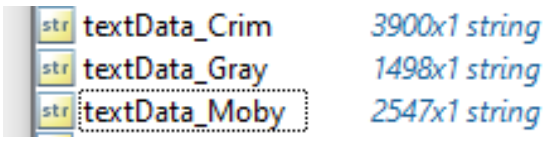
```
subtrees = findElement(tree, selector);
textData_Moby = extractHTMLText(subtrees);
```

The procedure above applies to *Moby Dick;* the user may proceed similarly with the other two novels, as the procedure is identical to the one outlined above. In the sequel, we denote the text data obtained for *Crime and Punishment* and *Dorian Gray* as *textData_Crim* and *textData_Gray*, respectively.

As the reader will have noticed, the procedure outlined just now incorporates the entire text from each book, including portions of text such as indexes and, for instance, the translator's preface for *Crime and Punishment*. These text data contribute nothing to the upcoming text mining procedure and can be easily removed via MATLAB's variable editor. To exclude the index and Translator's Preface from the *Crime and Punishment* string, for example, double-click the *textData_Crim* variable in the workspace to open the variable editor; then, highlight rows 1 to 64, right-click, and choose *Delete Row(s)*. Follow the same procedure with *textData_Moby*. The *Dorian Gray* string variable *textData_Gray* has no unwanted index or translator's preface.



Once the three texts have been processed, the workspace should be populated with a 3900×1 string *textData_Crim*, a 2547×1 string *textData_Moby*, and a 1498×1 string *textData_Gray*, as shown to the side.



As a cautionary step, it may be warranted to remove any empty string segments from the string arrays we've constructed. To do so, we apply the following command to each of the three *textData* objects.

```
textData_Moby = textData_Moby(~any(cellfun('isempty',
textData_Moby),2),:);
textData_Crim = textData_Crim(~any(cellfun('isempty',
textData_Crim),2),:);
textData_Gray = textData_Gray(~any(cellfun('isempty',
textData_Gray),2),:);
```

In the next step, we assemble a table containing the paragraphs in one column and the name of the book they belong to in a second column. We first add a second column to the *textData* variables containing the name of the book that corresponds to each paragraph:

```
textData_Crim(:,2) = 'CRIME AND PUNISHMENT';
textData_Moby(:,2) = 'MOBY DICK';
textData_Gray(:,2) = 'DORIAN GRAY';
```

In the next step, we assemble a table containing the text segment (which we call a *document*) in one column and the name of the book (variable *book*) in the other:

```
dataset_prelim.book = [textData_Moby(:,2); textData_Crim(:,2);
textData_Gray(:,2)];
dataset_prelim.text = [textData_Moby(:,1); textData_Crim(:,1);
textData_Gray(:,1)];
dataset = struct2table(dataset_prelim);
```

Then, we reset the random number generator and randomize the data.

```
% Resets the random number generator
rng('default');
dataset = dataset(randperm(height(dataset)),:);
```

Then, we preprocess *dataset* by tokenizing it, erasing punctuation, and lowercasing the text; the new variable is named *docs*; we also generate a variable *labels* for the name of the book that corresponds to each document, a variable *nbooks* containing the number of books, and a variable *books* containing the names of the three novels.

```
% Preprocesses the data
docs = tokenizedDocument(dataset.text);
docs = erasePunctuation(docs);
docs = lower(docs);
labels = dataset.book;
nbooks = 3;
books = ['MOBY DICK', 'CRIME AND PUNISHMENT', 'DORIAN GRAY'];
```

Then, we separate *docs* into three sets: a test set containing 1000 documents, a development set containing 1000 documents, and a train set containing the remaining (=5810) samples.

```
% Prepares the test set partition (1000 samples)
tstidx = 1:1000; %Defines the index range
ntst = length(tstidx); %Sixe of the test set
tstdocs = docs(tstidx); %Test set documents
tstlbls = labels(tstidx); %Test set labels

% Prepares the development set partition (1000 samples)
devidx = 1001:2000;
ndev = length(devidx);
devdocs = docs(devidx);
devlbls = labels(devidx);

% Prepares the train set partition (5810 samples)
trnidx = 2001:length(docs);
ntrn = length(trnidx);
trndocs = docs(trnidx);
trnlbls = labels(trnidx);
```

Next, we use the available data to create a bag-of-words model for the dataset. The BoW is created on the basis of the train set partition. In creating the BoW object, we remove infrequent words (in this case, words appearing in the train set 5 times or less) and frequent words (in this case, words ranking in place 1 to 40 in terms of No. of occurrences).

```
trnbow = bagOfWords(trndocs); %Computes bag-of-words model
%Removes infrequent words (counts <= 5)
trnbow = removeInfrequentWords(trnbow, 5);
%Removes 40 most frequent words
frqwds = topkwords(trnbow,40).Word;
trnbow = removeWords(trnbow, frqwds);
vocab_bow = trnbow.Vocabulary; %Stores the vocabulary
vsize_bow = length(vocab_bow); %Vocabulary size
```

Then, we use the bag-of-words model created above to compute TF-IDF matrices for all three partitions in the dataset:

```
%Computes TF-IDF matrices for train, development, and test
trntfidf = tfidf(trnbow);
tsttfidf = tfidf(trnbow, tstdocs);
devtfidf = tfidf(trnbow, devdocs);
```

Before we proceed, it is appropriate to save the variables we've generated thus far; in the following code snippet, we save the pertaining variables in a file *txtMining.mat*:

```
save textMining books nbooks dataset docs vocab_bow vsize_bow
ntst tstdocs tstlbls tsttfidf ndev devtfidf devdocs devlbls ntrn
trndocs trnlbls trnbow trntfidf
```

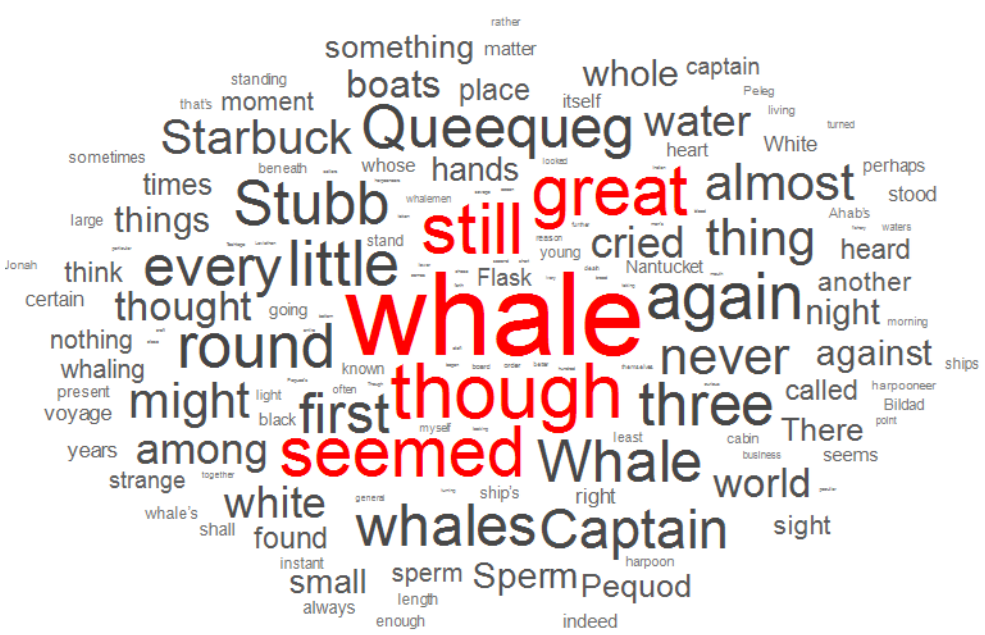
▶ Problem 3 – Keyword Extraction

Word clouds constitute the most intuitive tool for deriving important words in a text. The MATLAB function *wordcloud* uses word frequency counts to create a word cloud from a collection of strings. Let us create a word cloud for the *Moby Dick* text data. We begin by creating a tokenized document variable from *textData_Moby* and removing short words (in the present case, words with less than 5 characters) and stop words.

```
%Creates an array of tokenized documents
tokdocs = tokenizedDocument(textData_Moby(:,1));
vocab_Moby = tokdocs.Vocabulary; %Extracts the vocabulary
%Identifies short words (words with less than 5 characters)
shortWords = vocab(strlength(vocab_Moby));
%Removes short words and stop words
prepDocs = removeWords(tokdocs, union(shortWords, stopWords));
```

Next, the command *wordcloud* can be used to create the eponymous plot based on the list of most frequent words excluding short words and stop words. The pertaining code is shown below.

```
hf = figure(1);
set(hf, 'Color', [1,1,1], 'Name', 'Word Cloud of Frequent Words');
wordcloud(prepDocs, 'Shape', 'oval', 'HighlightColor', 'red', ...
    'MaxDisplayWords', 150, 'SizePower', 0.2);
```



As shown, the sizes of the words in the word cloud are scaled according to their frequency in the text; argument *SizePower* controls the scaling pattern. The most prominent word in the cloud is *whale*, which is unsurprising.

There are several shortcomings in the word cloud constructed just now. For instance, doing away with words containing less than 5 characters may exclude important terms, such as *sea* or *ship*. Secondly, the code above processes simple words only; phrases such as *great whale* and other multiword constructs are broken down into individual words. Some of these shortcomings can be addressed with two of MATLAB's keyword extraction algorithms, RAKE and TextRank, which we briefly introduce next.

The RAKE algorithm is implemented in the Text Analytics Toolbox™ function *rakeKeywords*. Let us apply this command to the *Moby Dick* text data and count the number of keywords extracted using *height*:

```
keywords_Moby = rakeKeywords(tokdocs);
keywords_Moby.Keyword = strip(join(keywords_Moby.Keyword));
height(keywords_Moby) %Number of keywords extracted
ans =
       66408
```

Since we have not specified the maximum number of keywords per document, all candidate keywords found in each document have been returned. We can see some candidate keywords in a certain document – say, document No. 10:

```
>> disp(keywords_Moby(keywords_Moby.DocumentNumber == 10, :))
                Keyword                      DocumentNumber    Score
    ____________________________________    ______________    _____

    "two orchard thieves entailed upon"          10             25
    "monied man enter heaven"                    10           15.5
    "man receives money"                         10            9.5
    "earnestly believe money"                    10              9
    "never pay passengers"                       10            7.5
    "passengers themselves"                      10            4.5
    "always go"                                  10              4
    "consign ourselves"                          10              4
    "earthly ills"                               10              4
```

A few advantages relatively to the frequency-based text processing are immediately apparent. First, RAKE generates keywords as phrases with a number of individual words ranging anywhere from 1 to 5 or even more. Secondly, RAKE does not necessarily reject all short words.

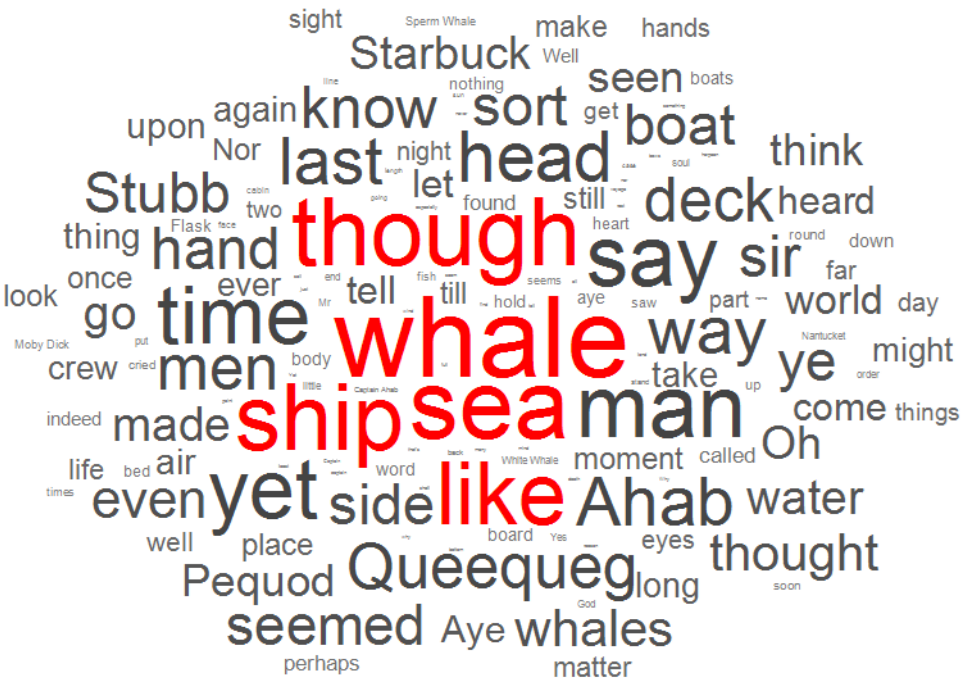
As important keywords are expected to be repeated across multiple sentences in the book, we can compute frequency counts for the candidate keywords over the entire dataset and rank them accordingly.

```
[uniqueKeywords, ~, uidx] = unique(keywords_Moby.Keyword);
keywordFreqs = hist(uidx, length(uniqueKeywords));
[rankedFreqs,ridx] = sort(keywordFreqs,'descend');
rankedKeywords = uniqueKeywords(ridx);
```

Then, we can use the top-ranked candidates and their frequencies to generate the word cloud.

```
hf = figure(3);
set(hf, 'Color', [1,1,1], 'Name', 'Word Cloud of RAKE Keywords');
wordcloud(rankedKeywords, rankedFreqs, 'HighlightColor', 'red', ...
    'MaxDisplayWords', 150, 'SizePower', 0.2);
```

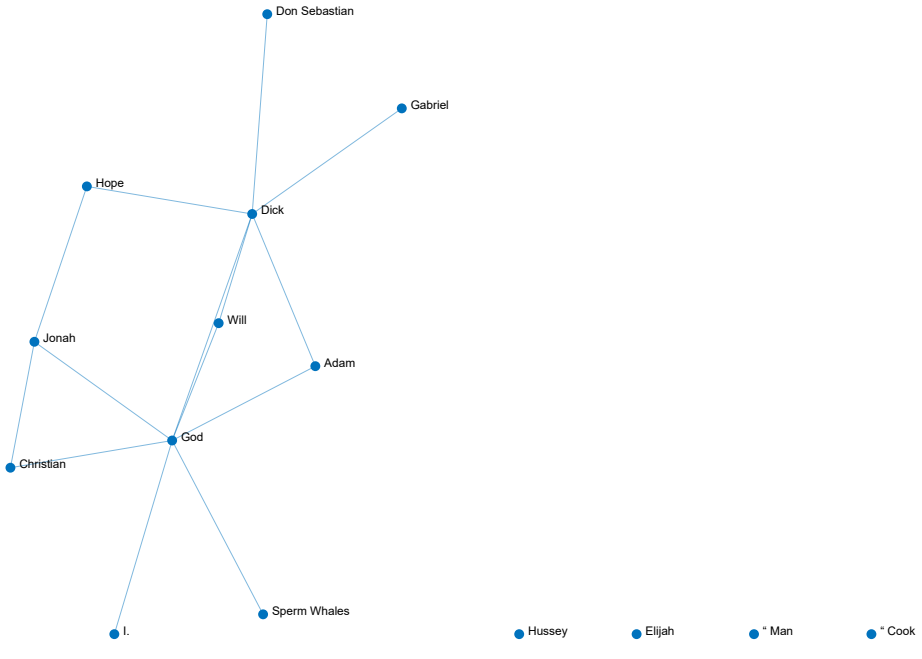
(*The word cloud is shown on the next page.*)



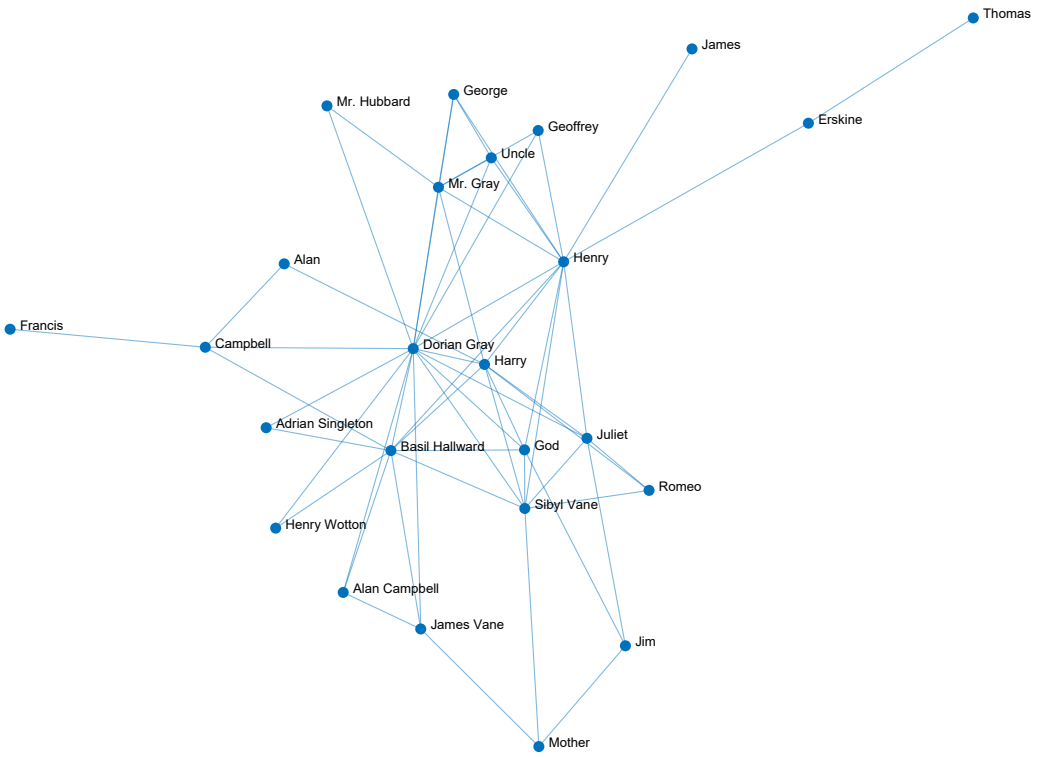
Some improvements are immediately apparent relatively to the frequency-based word cloud. Firstly, there is a greater prevalence of nouns, to the detriment of abstract words such as *first* or *still*. Secondly, short words such as *man*, *ship* and *sea*, which were rejected in the previous word cloud, appear prominently in the RAKE-based plot.

The TextRank keyword extraction tool, which is also part of the Text Analytics Toolbox™, uses the same classification procedure as RAKE, but in a graph-theoretical framework. Unidirected graphs are used to join words, and a famous algorithm called PageRank is used to compute node centrality. (Importantly, PageRank was the original algorithm used by Google's search engine when it was launched in 1998.) To illustrate the idea of centrality, we consider the names of the characters in the plot of *Moby Dick* and compute the co-occurrence matrix over the different sentences in the dataset. Such a co-occurrence matrix defines an undirected graph of characters for which node centrality scores can be computed.

```
newtokdocs = addEntityDetails(tokdocs);
details = tokenDetails(newtokdocs);
nonpersons = details.Token(details.Entity~='person');

%Creates a bag-of-words model
bow = bagOfWords(newtokdocs);
bow = removeWords(bow, nonpersons); %Removes non-persons
bow = removeInfrequentWords(bow,5); %Removes infrequent persons

%Computes the co-occurrence matrix and builds the graph
counts = bow.Counts;
cooccurrencemtx = counts.'*counts;
pgraph = graph(cooccurrencemtx, bow.Vocabulary, 'omitselfloops');

hf = figure(2); %Plots the graph
figname = 'Graph of most frequent characters in Moby Dick';
set(hf, 'Color', [1,1,1], 'Name', figname);
plot(pgraph); axis off
```



This graph network is rather uninteresting because *Moby Dick* doesn't have many recurring characters. Replacing the data with information from *Dorian Gray* and running the same code leads to the following graph plot, which is much more informative. Notice how many graphs are directed toward the 'Dorian Gray' graph, corroborating the importance of this character in the plot (see what I did there?). The plot is far from perfect, though, as there are separate graphs that actually refer to the same character (e.g., 'Alan' and 'Campbell'). Improvements on the quality of a character graph are investigated in Exercise 14,4-3 of Banchs (2021).



Once the graphs have been established, we may create a table *pgraph.Nodes* and compute different types of node centrality scores with the following commands:

```
pgraph.Nodes.Degree = centrality(pgraph, 'degree');
pgraph.Nodes.Closeness = centrality(pgraph, 'closeness');
pgraph.Nodes.Betweenness = centrality(pgraph, 'betweenness');
pgraph.Nodes.PageRank = centrality(pgraph, 'pagerank');
disp(pgraph.Nodes)
```

Name	Degree	Closeness	Betweenness	PageRank
{'Henry Wotton' }	2	0.018182	0	0.019125
{'Basil Hallward' }	10	0.025641	34.783	0.078448
{'Henry' }	12	0.027027	79.569	0.096304
{'Harry' }	9	0.02439	27.269	0.068591
{'Dorian Gray' }	16	0.030303	101.45	0.12159
{'Mr. Gray' }	7	0.022222	7.4286	0.055304
{'God' }	6	0.022727	10.5	0.046527
{'Uncle' }	4	0.02	0	0.033011
{'George' }	4	0.02	0	0.033011
{'Erskine' }	2	0.017241	23	0.028149
{'Thomas' }	1	0.012346	0	0.017977
{'Sibyl Vane' }	8	0.02381	20.84	0.060705
{'Romeo' }	3	0.016667	0	0.025598
{'Juliet' }	6	0.022727	12.79	0.04711
{'Mother' }	3	0.016667	2.0833	0.027402
{'James' }	1	0.016667	0	0.012824
{'Jim' }	3	0.016393	1.2	0.027028
{'James Vane' }	4	0.019608	6	0.034375
{'Mr. Hubbard' }	2	0.018182	0	0.019172
{'Adrian Singleton'}	2	0.018182	0	0.019125
{'Francis' }	1	0.013514	0	0.014607
{'Alan Campbell' }	3	0.018868	0	0.026427
{'Campbell' }	4	0.019608	27.083	0.040512
{'Alan' }	2	0.016667	1	0.021083
{'Geoffrey' }	3	0.019608	0	0.025996

As can be seen, for all flavors of character relatedness listed in the table, Dorian Gray is the most important entity in the text data we've fed to MATLAB. Henry (Wotton) is quite relevant, too.

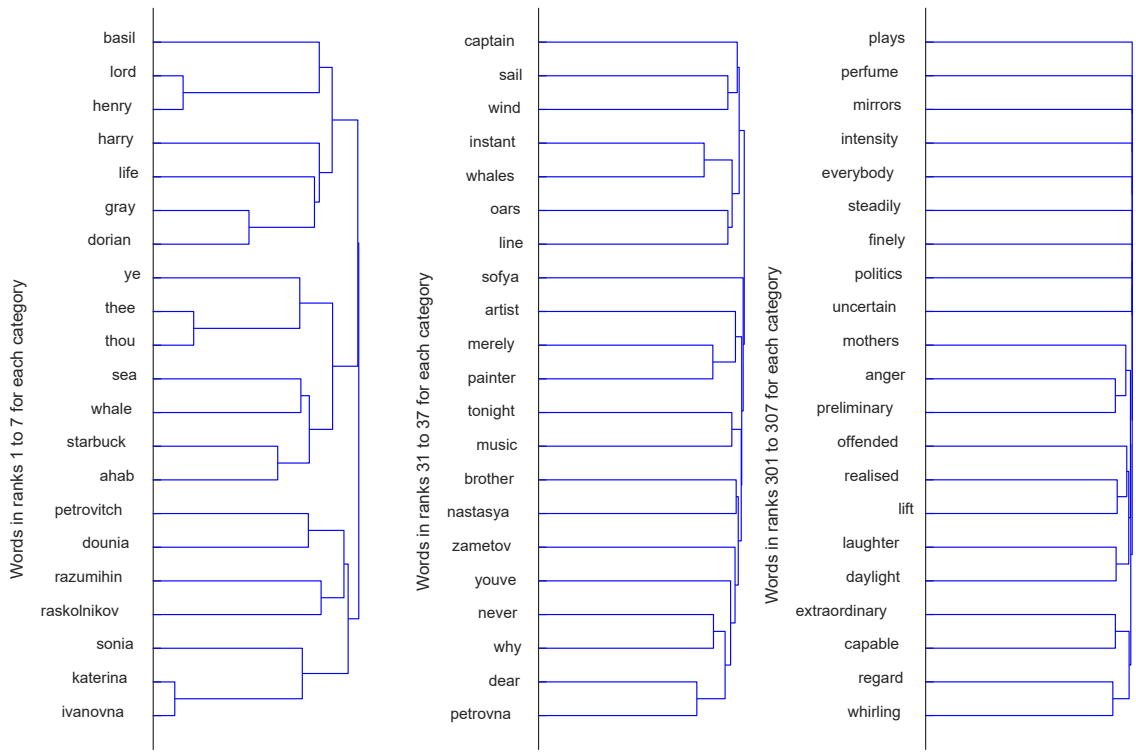
Another way to search for the most representative words in a data collection is through a geometrical approach. In vector space, the dimensions of document vectors correspond to the vocabulary terms in the document collection, and the distance scores used to assess similarities among document vectors rely on word co-occurrences and distributions across documents. In a geometrical keyword exploration framework, we first compute the average document vector for each of the three categories (books) in our collection. We do this over the train set:

```
%Gets the 10 most relevant words in category 1 (Moby Dick)
[~,idx1] = sort(meanvect(1,:)-meanvect(2,:)-meanvect(3,:),
'descend');
disp(vocab(idx1(1:10)))
%Gets the 10 most relevant words in category 2 (Crime and
Punishment)
[~,idx2] = sort(meanvect(2,:)-meanvect(1,:)-meanvect(3,:),
'descend');
disp(vocab(idx2(1:10)))
%Gets the 10 most relevant words in category 3 (Dorian Gray)
[~,idx3] = sort(meanvect(3,:)-meanvect(1,:)-meanvect(2,:),
'descend');
disp(vocab(idx3(1:10)))
Columns 1 through 9
"ahab"    "thou"    "ye"    "whale"    "thee"    "sea"    "starbuck"
"aye"    "boat"
  Column 10
  "ship"

  Columns 1 through 6
"raskolnikov"    "sonia"    "dounia"    "razumihin"    "ivanovna"
"petrovitch"
  Columns 7 through 10
    "katerina"    "svidrigaã"    "thats"    "lov"

  Columns 1 through 8
"dorian"    "henry"    "lord"    "gray"    "harry"    "basil"
"life"    "sibyl"
  Columns 9 through 10
 "hallward"    "picture"
```

Geometrically, the vectors computed in the code snippet above constitute the centroid of their corresponding category's set of vectors and, accordingly, provide a vector-based representation of the whole category. Banchs (2021) notes that we can think of these vectors as "average documents", which are the most representative documents of each category. To better illustrate the discriminative power of these sets of words with respect to other words in the vocabulary, we may construct dendrograms for different sets of words. In particular, we will be considering three groups of words: the set of most discriminative words (ranks 1 to 7) for each category, and two sets of less discriminative words (ranks 31 to 37 and ranks 301 to 307) for each category. In each case, a total amount of 21 words (7 from each category ranking) are to be considered. For these computations, we operate over the vector representations of words rather than documents:

```
%Selects the 7 most representative words from each category
words = [idx1(1:7), idx2(1:7), idx3(1:7)];
wordmtx = trntfidf(:,words)';
% Computes and plots the corresponding dendrogram
hf = figure(7);
ha = subplot(1,3,1);
set(hf, 'Color', [1,1,1], 'Name', 'Word dendrograms for different
word sets');
y = pdist(wordmtx, 'cosine');
z = linkage(y, 'average');
[h, t] = dendrogram(z, 0, 'labels', vocab(words), 'orientation',
'right');
temp = axis;
axis([0, 1, temp(3:4)]);
set(ha, 'Xcolor', [1,1,1]);
ylabel('Words in ranks 1 to 7 for each category');

%Selects words in ranks 101 to 107 from each category
words = [idx1(31:37), idx2(31:37), idx3(31:37)];
wordmtx = trntfidf(:,words)';
%Computes and plots the corresponding dendrogram
ha = subplot(1,3,2);
y = pdist(wordmtx, 'cosine');
```

```
z = linkage(y,'average');
[h, t] = dendrogram(z,0,'labels', vocab(words), 'orientation',
'right');
temp = axis;
axis([0, 1, temp(3:4)]);
set(ha, 'Xcolor', [1,1,1]);
ylabel('Words in ranks 31 to 37 for each category');

%Selects words in ranks 301 to 307 from each category
words = [idx1(301:307), idx2(301:307), idx3(301:307)];
wordmtx = trntfidf(:,words)';
%Computes and plots the corresponding dendrogram
ha = subplot(1,3,3);
y = pdist(wordmtx, 'cosine');
z = linkage(y, 'average');
[h, t] = dendrogram(z, 0, 'labels', vocab(words), 'orientation',
'right');
temp = axis;
axis([0, 1, temp(3:4)]);
set(ha, 'Xcolor', [1,1,1]);
ylabel('Words in ranks 301 to 307 for each category');
```



To resulting dendrograms are plotted above. As can be seen, when considering the 7 most relevant words for each category, a clear distinction among the three categories is noticeable (leftmost dendrogram); notice how the words from *Dorian Gray* are mostly clustered in the upper part of the dendrogram, whereas those from *Moby Dick* are clustered in the middle and those from *Crime* are clustered in the lower part. There are also mildly noticeable patterns in the central dendrogram, which refers to words ranked 31 to 37; indeed, the upper words *captain*, *sail* and *wind*, which are located in the upper region, call to mind *Moby Dick*; the words *artist*, *painter* and *music*, which are located in the middle, are related to *Dorian Gray*; the character names Nastasya and Zametov (the latter one a surname), which are located in the lower region, are related to *Crime*. However, when considering words in ranks 301 to 307, no distinction among the three categories is immediately apparent.

▶ Problem 4 – Document categorization

Document categorization goes beyond keyword extraction procedures. In addition to knowing important words within a text, we may also need to classify different texts within specific groups or categories. Document categorization algorithms designed for this purpose occur in the form of *unsupervised learning* (or simply *clustering*) and *supervised learning*. In unsupervised clustering, we automatically group objects in a given collection according to the similarities and differences of their salient features. Elements within each group or cluster are expected to exhibit substantial similarities among them, whereas elements across different groups or clusters are expected to exhibit large differences among them. The process is said to be *unsupervised* because no information about the categories coexisting in the collection is known beforehand. In supervised categorization, on the other hand, we have access to some sort of useful information about the

different categories in the data collection. In the present tutorial, we illustrate unsupervised learning through a technique called *k*-means clustering and unsupervised learning through the so-called *k*-nearest neighbors algorithm. We'll be using the same dataset prepared in Problem 2.

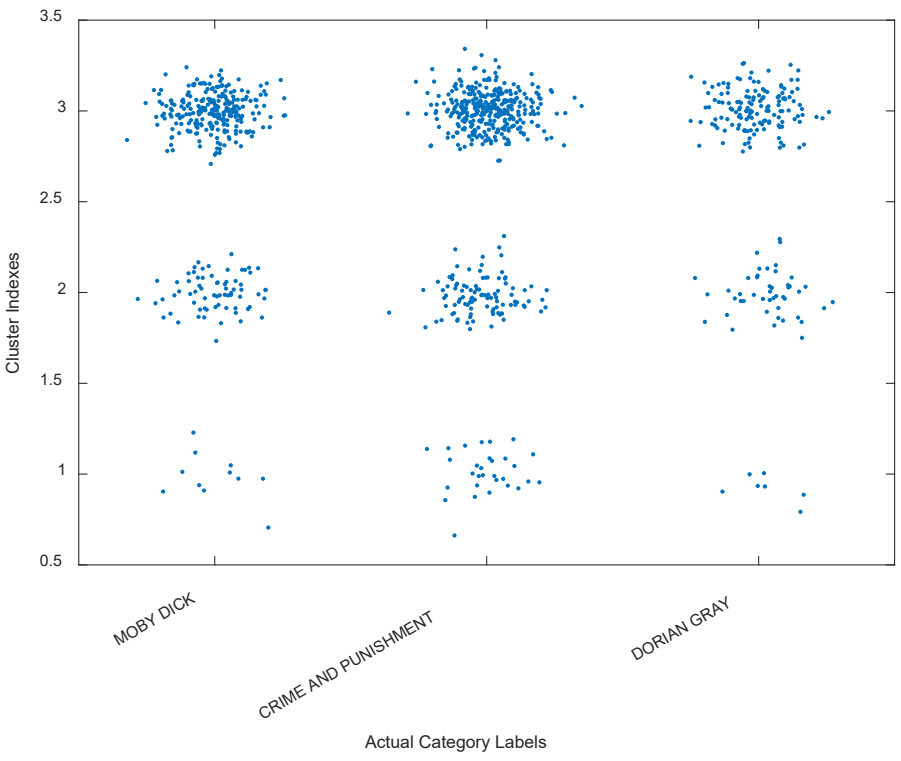
The MATLAB function *kmeans* implements the *k*-means clustering algorithm. The basic syntax is [*class*, *centroids*] = *kmeans*(*dataset*, *k*), where the input variable *dataset* is a matrix representation of the object collection, the rows being the observations (documents) and the columns being the variables (terms), and *k* is the desired number of clusters. The output variable *class* is a numeric array containing the index of the cluster to which each element in the collection has been assigned; the output variable *centroids* is a matrix containing the corresponding cluster centroid locations.

Crucially, the *k*-means algorithm requires the number of clusters *k* to be specified beforehand. This choice is highly subjective, but there are a few metrics, such as the Dunn index, that may help the modeler to make a more informed decision on this regard. Since we're working with three novels, an obvious choice on the number of clusters would be 3.

At this point, we should realize the problem of mapping the cluster indexes *indx* into the three books in the collection. As the clustering performed above is unsupervised in nature, we have no knowledge about the three books associated to the resulting clusters that relates the cluster indexes to the books. In order to proceed to evaluate classification accuracy, we must infer this mapping. As pointed out by Banchs (2021), one way to infer the mapping between cluster indices and books is by creating a cross-plot of index × books for the samples in the test set. As we expect the resulting clusters to be mainly aligned with the three book categories, we should be able to observe a larger concentration of samples in those cases in which the cluster index and the book correspond to each other. We create the cross-plot by using the following procedure (following Banchs, we add a small amount of noise for visualization purposes):

```
hf = figure(4);
figtitle = 'Cross-Plot Between Cluster Indexes and Category Labels';
set(hf, 'Color', [1,1,1], 'Name', figtitle);
for n = 1:length(tstlbls) %Gets book labels of samples in the test
set
    nlbl(n,1) = find(tstlbls(n)==books);
end
plot(nlbl+randn(size(nlbl))/10, idxs+randn(size(idxs))/10, '.');
xlabel('Actual Category Labels');
ylabel('Cluster Indexes');
xticks([1,2,3]); xticklabels(books);
```

The resulting cross-plot is shown below. The figure is not nearly as suggestive as we hoped it'd be; cluster number 3 is quite dense for all three books, while cluster 1 is sparsely populated for all three novels.



As an alternative, we may seek a mapping between clusters and books via a brute-force exploration of the amount of overlap between clusters and books across all possible mappings. This is illustrated by the following code:

```
%Considers all possible mappings
permutations = perms([1,2,3]);

%Computes cluster-book overlaps for all possible mappings
for n = 1:size(permutations,1)
    temp = 0;
    for k = 1:nbooks
        clusters = idxs'==k;
        permutedBooks = tstlbls==books(permutations(n,k));
        temp = temp + sum(clusters & permutedBooks);
    end
    overlaps(n) = temp;
end
%Gets the best mapping (i.e., maximum overlap)
[~,best] = max(overlaps);
books(permutations(best,:))
```

```
ans =

  1×3 string array

    "DORIAN GRAY"    "MOBY DICK"    "CRIME AND PUNISHMENT"
```

As shown, per the brute force approach we should assign cluster 1 to *Dorian Gray*, cluster 2 to *Moby Dick*, and cluster 3 to *Crime and Punishment*.

Now that we have established the appropriate mapping between the generated cluster indexes and the actual categories in the collection, we may evaluate the quality of the performed categorization over the test set. The main metric in this regard is the *accuracy* of the clustering scheme, which is defined as the ratio of the percentage of successful categorizations to the total amount of elements being categorized:

$$\text{Accuracy} = \frac{\text{Correct cases}}{\text{All cases}} \times 100\%$$

To compute the accuracy, we apply the mapping established above to all cluster indexes *idxs* assigned to the samples in the test set:

```
predictions = books(permutations(best,idxs))';
```

In this way, the indexes can be directly compared to the array of test set labels *tstlbls*:

```
accuracy = sum(predictions==tstlbls)/ntst*100
accuracy =

   41.7000
```

As shown, the resulting accuracy is 41.7%, which is mediocre but still better than random selection. The *k*-means algorithm has exploited the implicit structure of the dataset for generating a partition that approximates to a good extent the actual categories in the dataset. Notice that, aside from the number of categories coexisting in the dataset, no previous knowledge about the nature of the dataset has been used.

A better comparison between the resulting partition and the original document categories can be achieved through confusion matrices. Similarly to the cross-plot obtained above, the confusion matrix provides useful information about the degree of overlap among different clusters and categories.

```
confusion_mtx = confusionmat(predictions, tstlbls, 'ORDER', books);
cmtx = array2table(confusion_mtx, 'VariableNames', string(books));
cmtx.('Classified as') = string(books)';
disp(cmtx)
```

MOBY DICK	CRIME AND PUNISHMENT	DORIAN GRAY	Classified as
70	102	46	"MOBY DICK"
250	340	144	"CRIME AND PUNISHMENT"
11	30	7	"DORIAN GRAY"

Entries in the main diagonal refer to the number of documents that were classified correctly. For example, 340 documents from *Crime and Punishment* were classified correctly, whereas 250 were mistakenly classified in *Moby Dick* and another 144 were interpreted to belong to *Dorian Gray*.

Next, we turn to an example of *supervised* classification: the *k*-nearest neighbors or *knn* algorithm. This method is a remarkably simple but robust classification algorithm that, similarly to *k*-means clustering, operates over a vector space model. In the present application, we first train the *knn* model using the train set and evaluate it over the test set. Before the model can be trained, however, we want to select an optimal value for *k*. For this purpose, we use the development set and the commands *fitcknn* and *predict* for the training and inference tasks, respectively.

```
kvals = 3:10;
for k = 1:length(kvals)
    knn_model = fitcknn(trntfidf, trnlbls, 'NumNeighbors',
kvals(k));
    predictions = predict(knn_model, devtfidf);
    accuracy(k) = sum(devlbls==predictions)/ndev*100;
end
[maxaccuracy, idxoptim] = max(accuracy);
koptim = kvals(idxoptim); %Optimum value of k
fprintf('koptim = %d, maxacc = %5.2f\n', koptim, maxaccuracy)
```

```
>> koptim = 3, maxacc = 64.1
```

Once an optimal value for the parameter *k* has been established, we may train the *knn* algorithm with the train set clusters and evaluate its performance over the test set. As before, we use functions *fitcknn* and *predict* for the training and inference tasks, respectively:

```
%Trains a new knn model over the train set with k = koptim
knn_model = fitcknn(trntfidf, trnlbls, 'NumNeighbors', koptim);
%Computes accuracy of the generated model over the test set
predictions = predict(knn_model, tsttfidf);
accuracy = sum(predictions==categorical(tstlbls))/ntst*100
```

The lattermost code outputs *accuracy* = 62.2, which is significantly better than the value of 41.7 obtained with *k*-means clustering.

Using the same two algorithms we've employed in this part of the tutorial, Banchs (2021) obtained substantially greater accuracy values. The reason, possibly, is that Banchs worked with much larger documents, segmenting his data into large chapters instead of individual paragraphs, as we have done herein. Classification algorithms such as *k*-means or *k* nearest neighbors afford more efficient results when handling few, large chunks of text data than when handling many, short chunks of text data. Banchs goes on to show that multilayer perceptron (MLP), an algorithm based on artificial neural networks (ANNs), may yield better accuracy than either *k*-means of *knn* classification. This algorithm is beyond the scope of the present tutorial; the interested reader is referred to Banchs (2021).

REFERENCE

- BANCHS, R.E. (2021). *Text Mining with MATLAB*. 2nd edition. Berlin/Heidelberg: Springer.