Group_Project_2_Template

November 20, 2023

1 Group Project #2

Team Members: (Name and Identikey) * Member 1: Daniel Knopp * Member 2: Kenzie Hensel * Member 3: Ryan Hensel

Release Date: Nov 16th 1:30 pm

Due Date: Nov 29th 11:59 pm (Canvas)

Book selection: For this project, use the link to let the class know the books you will use. Do not choose a book that another group has already selected. https://docs.google.com/spreadsheets/d/11xpqzxlpKQHOBp-OPY7N9deE3UgdJxIB8LxysJtbp7M/edit?usp=sharing

Two groups cannot use the same book. A different book must be found if your first choice is taken.

Q1 Technical Book Selected: * Title: Python for Beginners * Author(s): Kuldeep Singh Kaswan, Jagjit Singh Dhatterwal, and B Balamurugan

Q2 Novel for letter/words evaluations: * Title: The Adventures of Sherlock Holmes * Author(s): Sir Arthur Ignatios Conan Doyle * Genre: Mystery, Detective Fiction, Crime Fiction, etc.

You will upload at least three files 1. Report (PDF file). Formal report with your responses for Q1 and Q2 2. Python Code (Jupyter Notebook) 3. .txt file of your novel book.

1.1 Q1 (40 points) Book - Table of Content

You have been tasked with creating a table of contents for a Data Science technical book. You are required to use a tree structure to organize your data hierarchically. Additionally, you must develop at least two functions, one to insert chapters or subchapter titles and another to print the table of contents.

You can find a technical book (use amazon.com or any other source). There are no restrictions, but the book must have at least three levels for a hierarchical view/analysis.

Create classes and functions to add chapters/subchapters & print TOC (Table of Contents).

You are free to create either a funcion *addChapter(node, title)* or *addLevelChapter(bookNode,level,title)* where level is a natural number from 0 to n that represent the heigh for a tree. Below is a sample case using **AddChapter** function:

```
myBook=Book("Book Title")
ch1=addChapter(myBook,"Title Chapter One")
```

```
ch11=addChapter(ch1,"Title of subchapter")
addChapter(ch11,"Title of subchapter")
addChapter(ch11,"Title of subchapter")

ch12=addChapter(ch1,"Title of subchapter")
addChapter(ch12,"Title of subchapter")
addChapter(ch12,"Title of subchapter")
addChapter(ch12,"Title of subchapter")

ch13=addChapter(ch1,"Title of subchapter")
addChapter(ch13,"Title of subchapter")
addChapter(ch13,"Title of subchapter")
addChapter(ch13,"Title of subchapter")

ch2=addChapter(myBook,"Title Chapter Two")
addChapter(ch2,"Title of subchapter")

ch22=addChapter(ch2,"Title of subchapter")
addChapter(ch22,"Title of subchapter")
addChapter(ch22,"Title of subchapter")

ch23=addChapter(ch2,"Title of subchapter")
addChapter(ch23,"Title of subchapter")
addChapter(ch23,"Title of subchapter")
addChapter(ch23,"Title of subchapter")
addChapter(ch23,"Title of subchapter")
```

Note: It's not permissible to include the chapter or subchapter number in the title's text. Instead, the print function will calculate and provide them. Furthermore, you should add a tab or double space for each deep level as shown in the sample output.

Sample of output

```
printTableOfContent(myBook)
```

Table of Contents

```
Book Title
1: Title Chapter One.....................................
  1.1: Title of subchapter..............................
    1.1.1: Title of subchapter..........................
    1.1.2: Title of subchapter..........................
  1.2: Title of subchapter..............................
    1.2.1: Title of subchapter..........................
    1.2.2: Title of subchapter..........................
    1.2.3: Title of subchapter..........................
  1.3: Title of subchapter..............................
    1.3.1: Title of subchapter..........................
    1.3.2: Title of subchapter..........................
    1.3.3: Title of subchapter..........................
```

```
2: Title Chapter Two.....................................
  2.1: Title of subchapter..............................
  2.2: Title of subchapter..............................
    2.2.1: Title of subchapter..........................
    2.2.2: Title of subchapter..........................
  2.3: Title of subchapter..............................
    2.3.1: Title of subchapter..........................
    2.3.2: Title of subchapter..........................
    2.3.3: Title of subchapter..........................
    2.3.4: Title of subchapter..........................
```

Tips/Notes:

- You have the freedom to pick up any book (Data Science book)
- You can create the most appropriate classes and functions using different data structures to solve this problem. Possible cases:
  - Case 1: Node is the main structure (one class), and each node has a list/array of nodes representing the chapters/subchapters (child nodes). Each note will have a text representing the title/subtitle (no numbers are included as a reference).
  - Case 2: Book as the main structure with internal class for chapters/subchapters with node and internal functions/methods/attributes to add chapters/subchapters and print the table of contents is ok, too.
- Technical books may have several chapters, but for this project, include only the first ten chapters with five subchapters, and five sub-subchapters as a minimal requirement. (you are free to add more real data)

Hint for the Table of Contents:

- The way you traverse the tree will provide the correct output for the table of contents. Review: PreOrder, InOrder, PostOrder, BFS, and DFS.
- You can approach the printTableOfContent function in three steps:
  - Step 1: Simple TOC with just titles.
  - Step 2: TOC with relative blank/tabs space to produce the right indentation.
  - Step 3: TOC indentation and numbering.

Report 1. Provide a description of your approach, data structure used and method to traverse the three (if apply) 2. Provide the code for your classes/functions with comments to make it clear for a reader 3. Provide a test code to validate your solution.

Description of Methodology

For out approach, we converted the PDF file into a TXT file and manually cleaned up the TXT so that it only contained the numbered table of contents entries. Then, we loaded in this TXT file and parsed it to get the individual lines (one TOC entry per line). We cleaned up each line to only contain the information we want and parsed the string using regular expression to extract the chapter title, the chapter id (numerical value associated with the chapter, subchapter, sub-subchapter, etc.), and the chapter level (depth of node within the tree). Using this information, we created new nodes for each main chapter, and for any sub or sub-sub, etc. chapter we first traverse the tree to find the correct parent and then add the new node. Since we already know the TOC

is in the correct order, we can use the fact that each children list will be populated also in order (makes finding the correct parent easy).

The tree data structure we chose to use was a general tree (each node can have many children). We also added additional attributes to each node so that the printing statement could be simplified. The chapter_id attribute was taken directly from the TXT file as-is and printed to represent the chapter.sub-chapter.sub-sub-chapter.etc for ease of reading. The chapter_level attribute was used to create the appropriate indentation for each line based on the depth of the node within the tree. The page number attribute was used just for reference to keep the correlation between chapter starts and page number. Finally, the chapter text is simply the title of the chapter or sub-chapter, etc.

For testing/validating our solution, we looped back over each line of the original file as well as the output lines from traversing our tree and compared them. We needed to make sure all the pieces we extracted from the original line existed in the correct lines and in the correct order. To do this, we used regular expressions to get the chapter id, chapter text, and page number from our final output string (from tree traversal) and then checked if those strings appeared in each line of the original text in the correct order. The result of this function returns True if everything matches or False if there are any errors.

[ ]:
```
# Create the Node class for the general tree
class Node:

    # Initialize the class, had attributes for chapter text, chapter id, and␣
 ↪chapter level and a list of children
    def __init__(self, text, id, level, page):
        self.chapter_text  = text
        self.chapter_id    = id
        self.chapter_level = level
        self.page_number   = page
        self.children = []

    # Methods to add children
    def add_child(self, obj):
        self.children.append(obj)

    # Method to get children based on list index
    def get_child(self, idx):
        return self.children[idx]

    # Method to print the tree using pre-order traversal
    def print_tree(self, out, max_char=100):
        txt = f'{"  " * (self.chapter_level+1)}{self.chapter_id} {self.
 ↪chapter_text} '
        out.append(f'{txt}{"." * (max_char - len(txt) - len(self.page_number) -␣
 ↪2)} {self.page_number}') # Adds indentation based on chapter level
        for child in self.children:
            child.print_tree(out)
```

[ ]:
```
# import statements
import re

# Read txt file
file = open("9781003202035_webpdf.txt", "r")
lines = file.readlines()
file.close()

# Initialize storage variables
prev_chapter    = 0
prev_subchapter = 0

# Create the root node with the Book's title
book = Node('Python for Beginners', '', -1, '0')

# Loop over all lines
for line in lines:

    # Remove any carriage returns from the end of the lines
    line = line.rstrip()

    # Get the page number at the end of the line
    page_number = re.findall('[0-9]*$', line)[0]

    # Use clean line to delete page numbers at the end of the line
    clean_line = re.sub('[ \t]*[0-9]*$', '', line)

    # Use Regular expression to find only the chapter level values
    chapter_id = re.findall('^[0-9.]+', clean_line)[0]

    # Remove chapter id values from the clean line
    clean_line = re.sub('^[0-9. ]+', '', clean_line)

    # Find the number of occurances of te pattern '.[0-9]' in the chapter id
    chapter_level = len(re.findall('\.[0-9]', chapter_id))

    # If the chapter level is 0 (main chapter)
    if chapter_level == 0:

        # Parse the chapter id to get the chapter number
        chapter_numbers = [int(chapter_id.split('.')[0])]

        # Add the chapter node to the book node as a child
        book.add_child(Node(clean_line, chapter_id, chapter_level, page_number))

    else:
```

```
        # Start with the root node (book)
        prev = book

        # Walk down the tree using chapter numbers in the chapter id, excluding␣
 ↪the last one (corresponds to the new node being added)
        for chapter_number in chapter_numbers[:-1]:

            # Get the correct child based on chapter number
            prev = prev.get_child(chapter_number-1)

        # After looping, will have the parent node for the new node, add the␣
 ↪new node as a child
        prev.add_child(Node(clean_line, chapter_id, chapter_level, page_number))

# Get the output from the print_tree method
out = []
book.print_tree(out)

# Print the output
for line in out:
    print(line)
```

```
 Python for Beginners
… 0
  1. Introduction to Python
… 1
    1.1 Introduction
… 1
    1.2 Software
…
1
    1.3 Development Tools
… 4
      1.3.1 Advanced Python Tools
… 5
      1.3.2 Web Scraping Python Tools
… 6
    1.4 learning about the Python Compiler
… 7
    1.5 Python History
… 8
    1.6 Python Installation
… 10
      1.6.1 Step 1: Select Version of Python to Install
… 10
      1.6.2 Step 2: Download Python Executable Installer
… 10
```

```
    13.1 Introduction
… 349
    13.2 Exception Examples
… 349
    13.3 Common Gateway Interfaces
… 358
    13.4 Database Access in Python
… 366
      13.4.1 What is MySQLdb?
… 366
      13.4.2 Database Connection
… 368
    13.5 The Read Operation
… 371
    13.6 Python Multithreaded Programming
… 379
      13.6.1 Starting a New Thread
… 379
      13.6.2 The Threading Module
… 380
      13.6.3 Thread Module
… 381
      13.6.4 Priority Multithreaded Queue
… 383
    13.7 Networking in Python
… 383
      13.7.1 What Are Sockets?
… 383
      13.7.2 The Socket Module
… 383
    13.8 Conclusion
… 387
```

[ ]:
```python
# Function to check validity of TOC
def is_valid_toc(lines, out):

    # Validate the the tree is correct by comparing the output to the original␣
 ↪text file
    for i, line in enumerate(lines):

        # Get chapter id from output
        chapter_id = re.findall('[0-9.]+', out[i+1])[0]

        # Get chapter text from output
        chapter_text = re.sub(' *[0-9].* ', '', out[i+1])

        # Get the page number from output
```

```
        page_number = re.findall('[0-9]*$', out[i+1])[0]

        # Check if the chapter id and text are in the original text file line
        if chapter_id not in line or chapter_text not in line or page_number␣
↪not in line:
            print(f'Error on line {i+1}')
            print(f'Chapter id: {chapter_id}')
            print(f'Chapter text: {chapter_text}')
            print(f'Page number: {page_number}')
            print(f'Line: {line}')
            return False

    # If no errors, return True
    return True
```

[ ]:
```
# Check if the TOC is valid
print(f'Is the TOC valid? {is_valid_toc(lines, out)}')
```

```
Is the TOC valid? True
```

1.2 Q2 (60 points) Counting words of a novel book

You can find a novel free of copyright and then review the word/letter structure in detail. You can use different data structures, but you will probably use strings, arrays, sets, and dictionaries (maybe trees).

You need to search for the (novel) book you would like to use in this project. The requirements for the book are:

- The book must be written in English
- The book should have more than 50,000 words
- Ideally, you need to find or convert to a .txt file format containing only text. If you have the book in PDF format, you can use an online website to convert it to a TXT file. https://pdfsimpli.com/, https://www.ilovepdf.com/

You need to decide the right Data Structure to solve the questions. However, you **cannot use NLTK** (Natural Language Toolkit) or other libraries to manage text (we will use them in other courses). You can use a built-in function for strings, stacks, dictionaries, queues, arrays, and sets.

Before starting the analysis, removing punctuation, special characters, and stopwords is essential.

What do you need to do (Report): 1. Describe your approach to your solution. Justify your decision about the data structures selected and the algorithms used (if applicable). Discuss the complexity in a BIG O Notation when applied. 2. Identify the distribution of each letter from a to z, including lower and upper cases. For example, A has the same treatment as a. ('A' is equal to 'a'). Provide frequencies in a summary table and a chart. 3. Remove stopwords (if you want, you can use the NLTK library to get the list of stopwords, nothing else). 4. Count the words and present the top 40 words used in the text. Include a word cloud in your report 5. Analyze bigrams (two words in sequence) and show the top 20. Include a word cloud in your report. https://en.wikipedia.org/wiki/Bigram 6. Analyze trigrams (three words in sequence) and

present the top 20. Include a word cloud in your report. https://en.wikipedia.org/wiki/Trigram 7. Finally, provide a written analysis of the data you found in points 2, 4, 5, and 6.

```
# Python Code
import string

# Storing the sets of punctuation in variable result
result = string.punctuation

# Printing the punctuation values
print(result)
```

```
# Python Code
# REMOVING PUNCTUATION
# You may use the string.punctuation string or create your own
# You can create a better methong (more optimized for long text)


# initializing string
test_str = "DSA is fundamental : for ! Data Science ;"

# printing original string
print("The original string is : " + test_str)

# initializing punctuations string
punc = '''!()-[]{};:'"\,<>./?@#$%^&*_~'''

# Removing punctuations in string
# Using loop + punctuation string
for ele in test_str:
    if ele in punc:
        test_str = test_str.replace(ele, "")

# printing result
print("The string after punctuation filter : " + test_str)
```

Tips: * As a group, you need to decide which book you want to evaluate. Select a book you are interested in because it will make more sense when processing the data. However, keep in mind that you cannot use a book that has already been taken or selected by another group.

- Make sure to remove any punctuation, special characters, and images from the text. You can also evaluate a list of prepositions to be removed but be careful when deciding which ones to keep and which ones to remove.

[ ]:
```
# Get only the standard stop words from nltk package
from nltk.corpus import stopwords
stop_words = stopwords.words('english')
```

Description of Methodology for Cleaning Text

First we downloaded the book as an ePub (Open Source Book Website) document and converted that to a text file. Then we did a little manual cleaning to remove all images, miscellaneous stuff at the beginning and end of the book (copywrite info, etc.), table of contents, and adjusted the chapter headings to follow a consistant format. After manually tidying the book text, we read the data into Python and started processing it.

The first step was to convert all letters to lower case, remove all carriage returns, and delete all punctuation, special characters, and numbers. Second, we cleaned up any extra whitespace (converted any occurances of 2 or more spaces to only a single space) and remove the chapter labels. Lastly, we removed all stop words using the list provided by the NLTK package.

[ ]:
```
# Read txt file
file = open('doyle-adventures-of-sherlock-holmes.txt', "r")
text = file.read()
file.close()

# Convert all text to lower case
text = text.lower()

# Remove all carriage returns
text = text.replace('\n', ' ')

# Remove all special characters and punctuation
to_delete = '''!()-[]{};:'"\,<>./?@#$%^&*_~ï¿½1234567890'''
for char in to_delete:
    text = text.replace(char, '')

# Remove all white space with more than 1 space
text = re.sub(' {2,}', ' ', text)

# Remove all chapter labels
text = re.sub(' chapter i* ', ' ', text)

# Remove all stop words
for word in stop_words:
    text = re.sub(f' {word} ', ' ', text)

# print(len(text))
# print(text)
```

Description of Methodology for Letter Counting

For counting the letter and their frequency of occurance, we decided to use a dictionary data structure. The algorithm was simple; loop over all letters, if the letter has not been added to the dictionary then initialize the entry with a frequency of 1, else if the letter has occured, increment the dictionary value (frequency) by 1. This procedure yields a dictionary whos keys correspond to the individual letters and whos values correspond to the frequency of occurance of those letters.

Next, for outputting a table of the occurances, we converted our dicionary to a format that was compatible with the Pandas package. We chose this becuse printing a datafram provides a easy

table format. We then sorted the dataframe in descending order by the frequency of occurance, printed the table, and plotted a histogram.

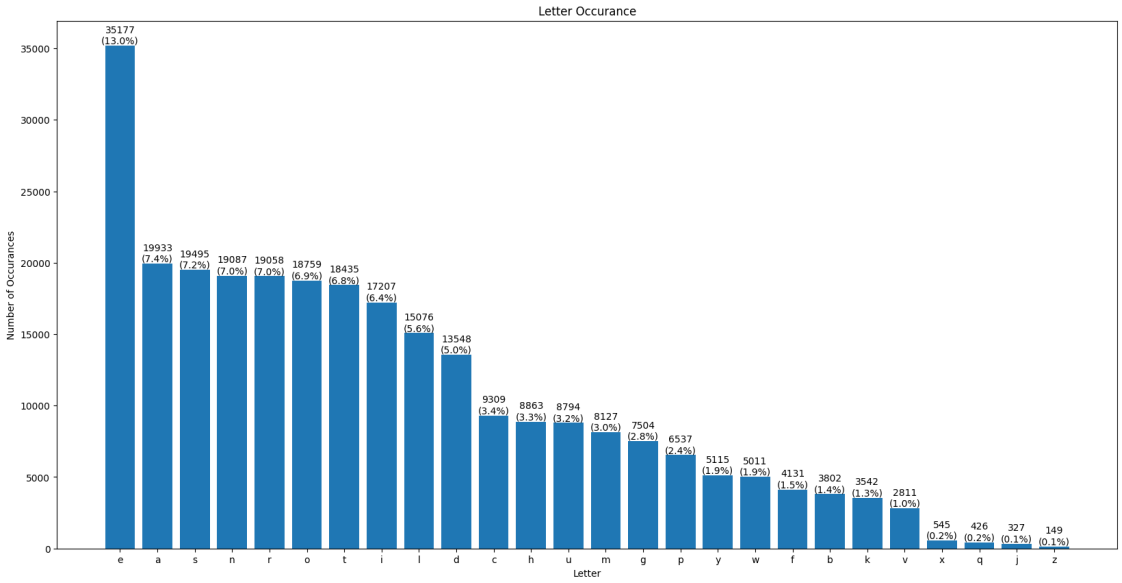
The time complexity for this algorithm is O(N) where N is the number of characters in the string. This is because we only loop over each character a single time when assigning or incrementing the value stored in the dictionary. Once we have created the dictionary, all remaining tasks are constant-time complexity and thus the worst-case complexity simplifies to O(N).

[ ]:
```
# Define a helper function to count the number of occurances of items in a list␣
↪(compatible with counting chars in strings and objects in lists)
def count_occurances(data):
    occurances = {}
    for item in data:
        if item in occurances:
            occurances[item] += 1
        else:
            occurances[item] = 1
    return occurances
```

[ ]:
```
import pandas as pd
import matplotlib.pyplot as plt

# Count the frequency of occurance of each unique letter, ecxluding spaces
letter_counts = count_occurances(text.replace(' ', ''))

# Create a new dictionary in the correct format to make a Pandas DataFrame
data = {'Letter': list(letter_counts.keys()), 'Frequency': list(letter_counts.
↪values())}

# Create a DataFrame from the dictionary
df = pd.DataFrame(data)

# Order the frequency table by frequency in descending order
df = df.sort_values(by='Frequency', ascending=False).reset_index(drop=True)

# Display the DataFrame table
print(df)

# plot a histogram of the letter frequency using matplotlib
plt.figure(figsize=(20,10))
plt.bar(df['Letter'], df['Frequency'])
plt.xlabel('Letter')
plt.ylabel('Number of Occurances')
plt.title('Letter Occurance')

# Add counts and percentage of total to the top of each bar
total_occurences = sum(df['Frequency'])
for i, v in enumerate(df['Frequency']):
```

```
    plt.text(i, v, f'{v}\n({v / total_occurences * 100:.1f}%)', ha='center',␣
 ↪va='bottom')

plt.show()
```

	Letter	Frequency
0	e	35177
1	a	19933
2	s	19495
3	n	19087
4	r	19058
5	o	18759
6	t	18435
7	i	17207
8	l	15076
9	d	13548
10	c	9309
11	h	8863
12	u	8794
13	m	8127
14	g	7504
15	p	6537
16	y	5115
17	w	5011
18	f	4131
19	b	3802
20	k	3542
21	v	2811
22	x	545
23	q	426
24	j	327
25	z	149



Analysis of Letter Counting

The distribution we found in the table and plot above makes sense according the the most-commonly and least-commonly used letters in the English language. According to Grammarly (website), the most commonly used letters in the English language are: e, t, a, i, o, n, s, h, and r - and the least commonly used letters are: j, q, x, and z. This matches nearly perfectly with our analysis. The only difference is the existance of l, d, and c before h in our book's occurance order. We think this makes sense becasue the words "said", "could", and "would" occur many times (as we'll show below) - this would account for an increased occurance of the letters l, d, an c over h. Also, our analysis matches up perfectly that the least common letter are j, q, x, and z (by a clear margin).

Description of Methodology for Word Counting

We used the same data structure for counting words as we did for counting letters (reused the helper function for counting occurances). The difference here though, is we needed an additional pre-processing step to split the text from a array of characters (string) into a list of words. We did this by using the space character as delimiter using the standard .split() method for strings to output a list of individual words. Then, to compute the frequency, we followed the exact same procedure as for the letter counting and used the helper function defined above. We also created a Pandas dataframe ans sorted by frequency to print only the top 40 most connon words.

To create the word cloud diagram, we imported the wordcloud package and fed in the dictionary that is output by our helper frequency counter function. This procudes a word cloud where the size of the word corresponds to the number of occurances (larger = more frequenc, smaller = less frequent).

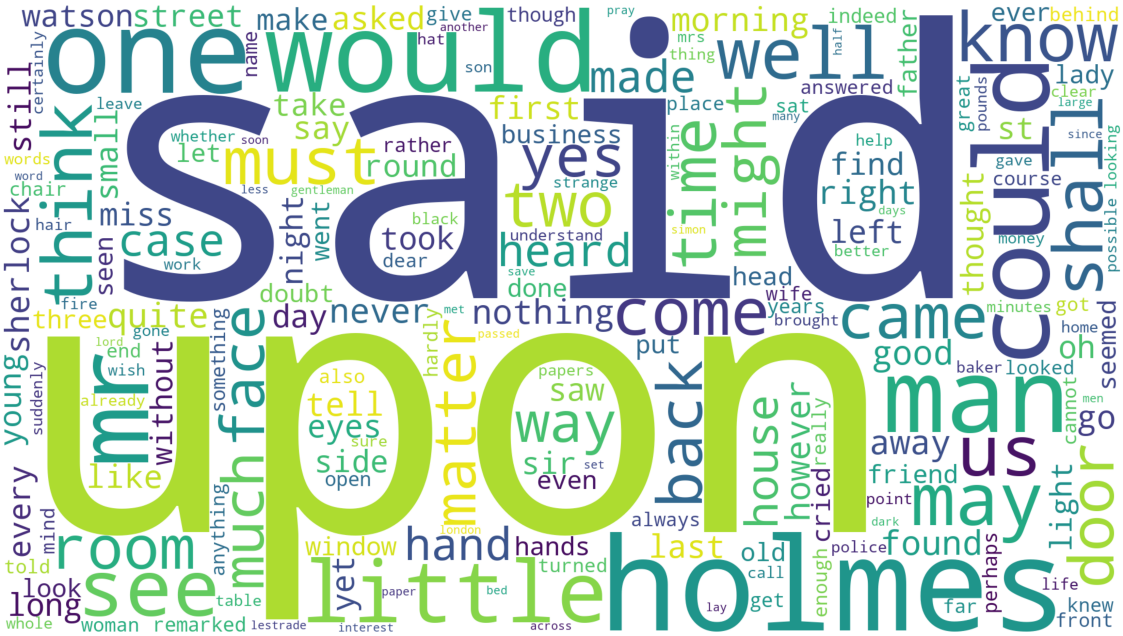
The time complexity of this algorithm is O(N + M) where M corresponds to the number of words and N corresponds to the number of characters. The reason for this is that the helper fucntion performs the same when looping over characters in a string as it does for words in a list and the resulting time complexity for this step is O(M) for the same reason as discussed above for counting letters. The additional pre-processing step of parsing the string into words takes O(N) because the

split() function must loop over all character in order to find the occurances of spaces to be used as the character for splitting words. Combined, this makes the time complexity O(N + M).

[ ]:
```
from wordcloud import WordCloud

# Get a list of all words in the text
words = text.split(' ')

# Count the frequency of occurance of each unique word
word_counts = count_occurances(words)

# Create a new dictionary in the correct format to make a Pandas DataFrame
data = {'Word': list(word_counts.keys()), 'Frequency': list(word_counts.
 ↪values())}

# Create a DataFrame from the dictionary
df = pd.DataFrame(data)

# Sort the datafram by frequency in descending order and reset the index for␣
 ↪the new order
df = df.sort_values(by='Frequency', ascending=False).reset_index(drop=True)

# Print the top 40 words
print(df.head(40))

# Create a WordCloud object
wordcloud = WordCloud(width=1920, height=1080, background_color='white').
 ↪generate_from_frequencies(word_counts)

# Plot the word cloud
plt.figure(figsize=(50, 25))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.show()
```

```
       Word  Frequency
0      said        486
1      upon        464
2    holmes        458
3       one        369
4     would        327
5       man        287
6     could        286
7        mr        274
8    little        269
9       see        228
10      may        197
11     well        185
```

12	us	182
13	think	174
14	know	169
15	shall	169
16	must	161
17	come	161
18	time	150
19	came	146
20	two	143
21	door	140
22	back	138
23	room	133
24	face	128
25	might	126
26	matter	125
27	yes	124
28	much	120
29	way	114
30	heard	113
31	house	112
32	hand	112
33	case	109
34	nothing	108
35	made	108
36	never	107
37	found	107
38	however	107
39	good	106



Analysis of Word Counting

The distribution show in the figure above makes sense because there is a lot of conversations in this book and thus it is logical for the word “said” to be the most common word as it nearly always follows a line of dialogue. Additionally proper nouns such as the name “Holmes” are very common as well for character that are prevelant in the book. Additionally, common words such as “could”, “would”, “upon”, and “one” make sense as they are very common when describing things in a novel.

Description of Methodology for Bigram Counting

We used the same data structure for counting bigrams as we did for counting letters (reused the helper function for counting occurances). The difference here though, is we needed an additional pre-processing step to split the text from a array of characters (string) into a list of bigrams. We did this by using a regular expression pattern to split the text into individual bigrams. To ensure we get all possible combination of 2-word bigrams, we then also removed the first word and recomputed the bigrams using the same regular expression. This makes sure that each word is counted in a bigram with the word prior and after. Then, to compute the frequency, we followed the exact same procedure as for the letter counting and used the helper function defined above. We also created a Pandas dataframe ans sorted by frequency to print only the top 40 most connon words.

To create the word cloud diagram, we imported the wordcloud package and fed in the dictionary that is output by our helper frequency counter function. This procudes a word cloud where the size of the word corresponds to the number of occurances (larger = more frequenc, smaller = less frequent).

The time complexity of this algorithm is O(N + 2M) -> O(N + M) where M corresponds to the number of bigrams (multiplied by 2 because we compute the bigrams twice) and N corresponds to the number of characters. The reason for this is that the helper fucntion performs the same when looping over characters in a string as it does for bigrams in a list and the resulting time complexity for this step is O(M) for the same reason as discussed above for counting letters. This step also occurs twice to make sure we get all possible combinations of 2 words. The additional pre-processing step of parsing the string into words takes O(N) because the split() function must loop over all character in order to find the occurances of spaces to be used as the character for splitting words. Combined, this makes the time complexity O(N + 2M) which simplifies to O(N + M) for the worst-case.

[ ]:
```
# Get all 2-word sequences in text starting from first word
two_word_sequences = re.findall('[a-z]+ [a-z]+', text)

# Add all 2-word sequences in text starting from second word
two_word_sequences = two_word_sequences + re.findall('[a-z]+ [a-z]+', re.
 ↪sub('^[a-z]+ ', '', text))

# Count the frequency of occurance of each unique 2-word sequence
two_word_counts = count_occurances(two_word_sequences)

# Create a new dictionary in the correct format to make a Pandas DataFrame
```

```
data = {'Sequence': list(two_word_counts.keys()), 'Frequency':␣
 ↪list(two_word_counts.values())}

# Create a DataFrame from the dictionary
df = pd.DataFrame(data)

# Sort the datafram by frequency in descending order and reset the index for␣
 ↪the new order
df = df.sort_values(by='Frequency', ascending=False).reset_index(drop=True)

# Print the top 20 2-word sequences
print(df.head(20))

# Create a WordCloud object
wordcloud = WordCloud(width=1920, height=1080, background_color='white').
 ↪generate_from_frequencies(two_word_counts)

# Plot the word cloud
plt.figure(figsize=(50, 25))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.show()
```

```
           Sequence  Frequency
0       said holmes        112
1   sherlock holmes         95
2         mr holmes         69
3          st simon         39
4         could see         31
5      baker street         29
6           lord st         28
7          st clair         23
8            let us         21
9        upon table         20
10       young lady         19
11          yes sir         17
12      miss hunter         17
13     hosmer angel         17
14      mr rucastle         17
15       neville st         16
16          one two         16
17      watson said         16
18           oh yes         15
19      miss stoner         15
```

Analysis of Bigram Counting

The distribution show in the figure above makes sense because Sherlock Holmes is the main character of the book and often says things. This seems to be why we often see the bigram “said holmes” and also references to the main character himself with “my holmes” and “sherlock holmes”. Many of the other common bigrams also reference proper nouns such as names or locations (i.e. “st clair”, “st simon”, “baker street”). Since there are so many conversations, it makese sense that so many proper nouns show here to identify the speakers.

Description of Methodology for Trigram Counting

We used the same data structure for counting trigrams as we did for counting letters (reused the helper function for counting occurances). The difference here though, is we needed an additional pre-processing step to split the text from a array of characters (string) into a list of trigrams. We did this by using a regular expression pattern to split the text into individual trigrams. To ensure we get all possible combination of 3-word trigrams, we then also removed the first word and recomputed the trigrams using the same regular expression - and then agian after removing the first and second words. This makes sure that each word is counted in a trigram with the 2 words prio, word before and after, and 2 words after. Then, to compute the frequency, we followed the exact same procedure as for the letter counting and used the helper function defined above. We also created a Pandas dataframe ans sorted by frequency to print only the top 40 most connon words.

To create the word cloud diagram, we imported the wordcloud package and fed in the dictionary that is output by our helper frequency counter function. This procudes a word cloud where the size of the word corresponds to the number of occurances (larger = more frequenc, smaller = less frequent).

The time complexity of this algorithm is O(N + 3M) -> O(N + M) where M corresponds to the

number of trigrams (multiplied by 3 because we compute the trigrams thrice) and N corresponds to the number of characters. The reason for this is that the helper fucntion performs the same when looping over characters in a string as it does for objects in a list and the resulting time complexity for this step is O(M) for the same reason as discussed above for counting letters. This step also occurs thrice to make sure we get all possible combinations of 3 words. The additional pre-processing step of parsing the string into words takes O(N) because the split() function must loop over all character in order to find the occurances of spaces to be used as the character for splitting words. Combined, this makes the time complexity O(N + 3M) which simplifies to O(N + M) for the worst-case.

[ ]:
```
# Get all 3-word sequences in text starting from first word
three_word_sequences = re.findall('[a-z]+ [a-z]+ [a-z]+', text)

# Add all 3-word sequences in text starting from second word
three_word_sequences = three_word_sequences + re.findall('[a-z]+ [a-z]+␣
 ↪[a-z]+', re.sub('^[a-z]+ ', '', text))

# Add all 3-word sequences in text starting from third word
three_word_sequences = three_word_sequences + re.findall('[a-z]+ [a-z]+␣
 ↪[a-z]+', re.sub('^[a-z]+ [a-z]+ ', '', text))

# Count the frequency of occurance of each unique 3-word sequence
three_word_counts = count_occurances(three_word_sequences)

# Create a new dictionary in the correct format to make a Pandas DataFrame
data = {'Sequence': list(three_word_counts.keys()), 'Frequency':␣
 ↪list(three_word_counts.values())}

# Create a DataFrame from the dictionary
df = pd.DataFrame(data)

# Sort the datafram by frequency in descending order and reset the index for␣
 ↪the new order
df = df.sort_values(by='Frequency', ascending=False).reset_index(drop=True)

# Print the top 20 2-word sequences
print(df.head(20))

# Create a WordCloud object
wordcloud = WordCloud(width=1920, height=1080, background_color='white').
 ↪generate_from_frequencies(three_word_counts)

# Plot the word cloud
plt.figure(figsize=(50, 25))
plt.imshow(wordcloud, interpolation='bilinear')
plt.axis('off')
plt.show()
```

	Sequence	Frequency
0	lord st simon	27
1	neville st clair	14
2	mr hosmer angel	13
3	mr sherlock holmes	11
4	mr neville st	9
5	said sherlock holmes	8
6	colonel lysander stark	7
7	sherlock holmes sat	7
8	sir george burnwell	7
9	mrs st clair	7
10	k k k	6
11	dr grimesby roylott	6
12	miss mary sutherland	6
13	mr jabez wilson	6
14	mr duncan ross	6
15	mr henry baker	6
16	watson said holmes	6
17	said holmes laughing	6
18	dear young lady	5
19	lady st simon	5

Analysis of Trigram Counting

The distribution show in the figure above makes sense because many of the proper nouns (proper titles) in this book contain 3 or more words in sequence. Some examples are “lord st simon”, “mr sherlock holmes”, and “mr hosmer angel”. There are also a number of trigrams that contin sherllock homes and a verb such as “said” or “sat” - this also makese sense because the main character will

be doing a lot of things in the book about himself.

An interesting confustion we had was we though the trigram “k k k” must have been an error from removing character, numbers, and stop words because we were not expecting this book to reference the Ku Klux Klan. After investigating the original text, we found that there was a specific case Sherlock Holmes was investigating that involved the “K. K. K.” initials. Since we removed punctuation, this showed up in our analysis as “k k k” and we were able to confirm that in-fact, the Ku Klux Klan was a part of the book and it is reasonable for it to show up here because any reference to them as an entity would involve these 3 initials being written in sequence.