SSIE500 Homework 2

Grant T. Aguinaldo

September 17, 2020

1 Introduction

As part of this homework assignment, an analysis to determine the distribution of letters was completed. In particular, this assignment sought to determine:

- The distribution of letters (count frequencies) in a text document;
- The distribution of transitions from one character to another in a text document.

The `Python 3.8` code that was generated for this project is included as Appendix A to this document. This document was created with Overleaf and uses `graphicx` to insert the figure images and `pdfpages` to embed the pdf document that contains the code used in this project. The code shown in the Appendix was generated using the Jupyter Notebook integrated development environment (IDE), rendered as a Tex file using functionality built into the Notebook interface, and then converted to a pdf. The resulting pdf was then included in this document using `pdfpages`.

1.1 Project Requirements

This project has several requirements that need to be met. A comparison of how each of the project requirements were met is included in **Table** 1. For clarity, the abbreviation "App" refers to the Appendix that is included in this document.

The code for this project is being maintained under version control using git.[1]

1.2 Data

The data for this project was obtained from the Project Gutenberg website. Project Gutenberg provides access to more than 60,000 free eBooks in a variety of formats. We decided to complete this assignment using the novel, Pride and

[1] https://github.com/grantaguinaldo/ssie/tree/master/ssie500/hw2

Table 1: Summary of Requirements

	Specification Section	Report Section
Prepare text document.	1	App. Cell 1
Read text document.	2(a)	App. Cell 2
Determine count frequency.	2(b)	App. Cell 7
Determine transition frequency.	2(c)	App. Cell 10
Visualize results.	2(d)	App. Cell 15
Generate 3,000 character string.	2(e)	App. Cell 12
Produce LaTeX report using `pdfpages`.	3	1

Prejudice by Jane Austen for two reasons. [2] First, Pride and Prejudice is one of the most commonly downloaded eBooks on the Project Gutenberg and second, this novel contains more than 750,000 characters and therefore, using this book should capture the normal variation of letters used in the English language.

1.2.1 Data Cleaning

For this project, we decided to only analyze the natural patterns between the letters found in Pride and Prejudice. That is, performed a number of cleaning tasks to remove the following characters prior to doing the analysis:

- Removed all punctuation from the dataset;
- Removed all characters that contain accent marks from the dataset;
- Converted all of the words in the dataset to lower case; and
- Removed all digits from the dataset.

For clarity, we would like to note that we did keep all of the white spaces in the dataset.

2 Analytical Approach

For this project, we largely relied on the `Pandas` library and used a combination of imperative and functional programming paradigms.

An example of an imperative paradigm used in this project includes the use of the `Pandas` method `load_csv()` that is used to load the `.txt` file into a `Pandas` dataframe. An example of a functional programming paradigm used in this project includes the creation and use of the `marvkov_sampler` function (see Cell 2 in the Appendix) that creates the random string found in **Figure** 3.

[2] https://www.gutenberg.org/ebooks/1342

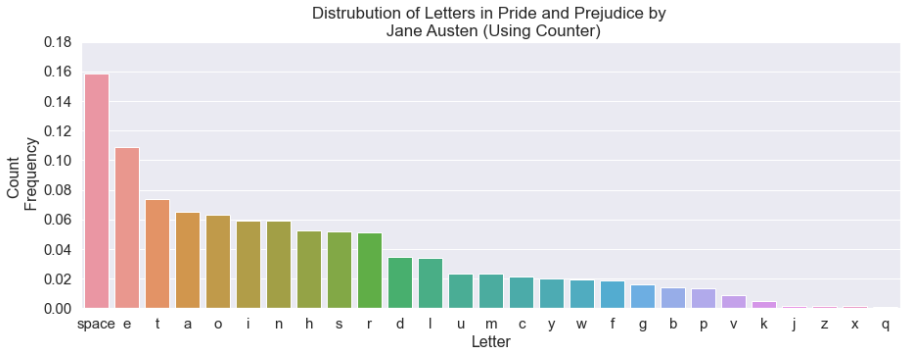


Figure 1: Distribution of Characters in Pride and Prejudice

3 Results

3.1 Most Common Letters

As part of this work, we determined the count frequency of the word in the text document in two different ways, [1] by using the built-in `Counter` class[3] and [2] by using the `count()` method [4] that is part of the `collections` library also within Python. As shown in Cell 6 in Appendix A, both methods produced the same results. Therefore, for rest of this analysis we only show the results by using the `Counter` class since computing the character occurrences required less code than the `count()` method.

As shown in **Figure** 1 the top five most common characters found in the text where 'space', 'e', 't', 'a', and 'o.' These five characters accounted for more than 41 % of all of the letters found in the data.

3.2 Most Common Transitions

As part of this project, we also determined the transition frequency for each of the words in the data set. For this portion of the work, we sought to break up each word in the text into two character segments, and then we determined the count frequency for all of the two character segments. We refer the reader to Cell 9 of the Appendix for the code used to generate these transition frequencies as well as to the table of all counts for the bigrams when considering the alphabet.

To visualize the Markov transition frequencies, we opted to use a heatmap as shown in **Figure** 2. As shown in **Figure** 2 the letter 'e' is a common transition letter in the text. That is, given a letter in the first position, the most common letter in the second position in the letter 'e.' Also, as can be seen in **Figure** 2, the two-letter sequences 'za', 've' and 'qu' are among the most common sequences in the data set.

[3] https://docs.python.org/3/library/collections.html
[4] https://docs.python.org/3.8/library/stdtypes.html?highlight=countstr.count

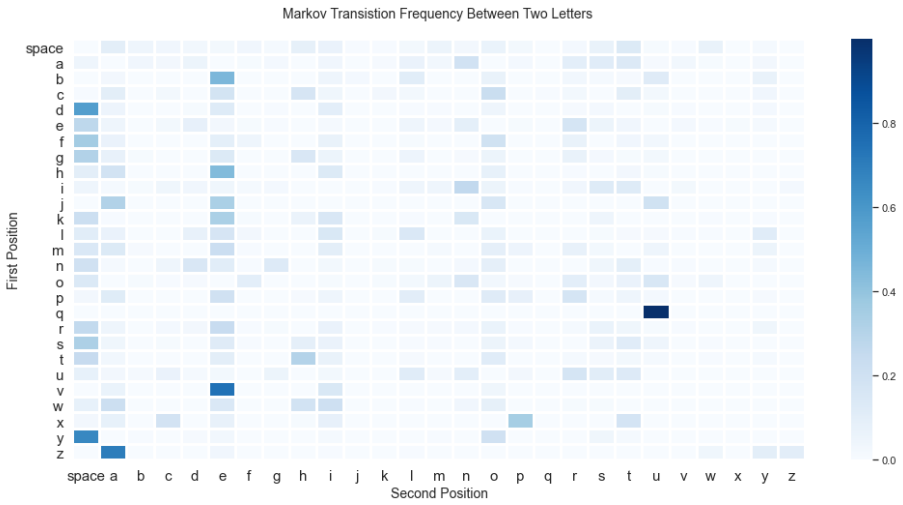


Figure 2: Heatmap of Markov Transition Frequencies

3.2.1 Markov Sampler

As part of this project, we also sought to generate a random text string using the most common Markov transition frequencies found in **Figure** 2. In particular, we used these transition frequencies and generated a 3,000 character sequence starting with the letter 't.' The resulting sequence in shown in **Figure** 3.

4 Discussion

This project sought to determine [1] the distribution of letters (count frequencies) in a text document and [2] the distribution of transitions from one character to another in the same text document using `Python 3.8`. As noted in 1.2 the text document we decided to use for this project was a long-form novel, called Pride and Prejudice by Jane Austen. We decided to use this book for two reasons, first, Pride and Prejudice is one of the most commonly downloaded eBooks on the Project Gutenberg and second, this novel contains more than 750,000 characters and therefore, using this book should capture the normal variation of letters used in the English language.

4.1 Most Common Letters in Data

In section 3.1 we noted that the top five most common characters found in the text where 'space', 'e', 't', 'a', and 'o.' Based on our observations, these five characters accounted for more than 41 % of all of the letters found in the data. These results are consistent with what was noted on the Wikipedia entry for

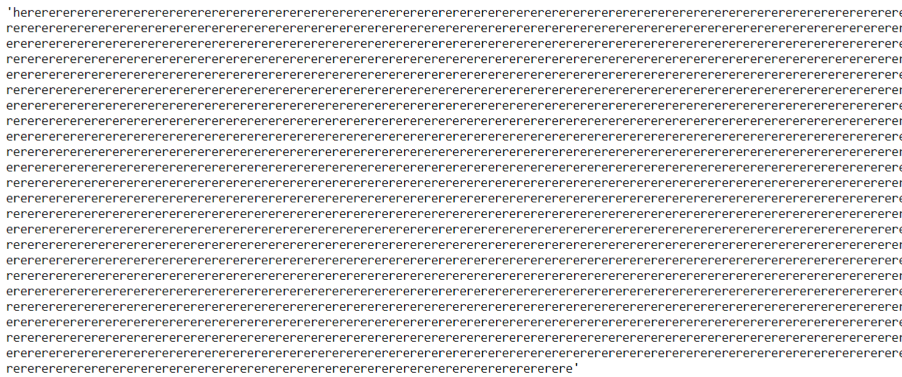


Figure 3: Random 3,000 Character String Generated from the Markov Transition Frequencies.

"Letter Frequency"[5] as well as other sources we found on the Internet.[6] [7]

A visualization of the distribution of letters/characters in the data is presented in **Figure** 1.

4.2 Most Common Letter Transitions

In section 3.2 we noted the results of the most common letter transitions. As shown in **Figure** 2 the letter 'e' is a common transition letter in the text. In other words, given a letter in the first position, the most common letter in the second position in the letter 'e.' Also, as can be seen in **Figure** 2, the two-letter sequences 'za', 've' and 'qu' are among the most common sequences observed in the data set. These results are different than what was noted on one Digraph Frequency chart that we found. We speculate that one reason for this difference is that the Digraph chart was generated using a sample size that was smaller than the data set used in this project.[8]

In passing, we like to note that we did initialize our Markov Sampler with other letters and we observed that the resulting strings did converge to the bigram "re" (results not shown). These results are consistent with our observation that the letter 'e' is a common transition letter in the text.

4.3 Future Work

Future work for this project can include building a weighted directed graph of the transition frequencies using the `networkx` package[9] in `Python`.

[5]https://en.wikipedia.org/wiki/Letter$_{f}$*requency*
[6]https://www3.nd.edu/ busiforc/handouts/cryptography/letterfrequencies.html
[7]http://pi.math.cornell.edu/ mec/2003-2004/cryptography/subs/frequencies.html
[8]http://pi.math.cornell.edu/ mec/2003-2004/cryptography/subs/digraphs.html
[9]https://networkx.github.io/

Appendix: Homework 2

September 11, 2020

1 SSIE 500 Homework 2 (Playing with Python)

Grant T. Aguinaldo

[1]:
```
# Imported Packages for the Project
import pandas as pd
import string
import numpy as np
from collections import Counter, OrderedDict
import matplotlib.pyplot as plt
import copy
import requests as r
import seaborn as sns
%matplotlib inline

# URL to the text file of Pride and Prejudice that was downloaded from http://
 ↪www.gutenberg.org/ebooks/1342
# This code fulfills the requirements outlined in 1 of the homework instructions.
url = 'https://raw.githubusercontent.com/grantaguinaldo/mcmc/master/
 ↪pride_prejudice.txt'
```

[2]:
```
# Set of User-Defined Functions

def load_data(url):
    '''
    This function downloads a text document from a url
    and does transforms the text into a python list that
    can be loaded into a Pandas DataFrame.

    This code fulfills the requirements outlined in 2(a) of the homework␣
 ↪instructions.
    '''
    string_punctuation = '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~‘êàé-’’“‘'
    url = url
    data = r.get(url)

    # This code fulfills the requirements outlined in 2(a) of the homework␣
 ↪instructions.
```

```
    f = data.text
    print('Text File Has Been Downloaded')
    print('---')
    print('This Text File Contains: {} Characters'.format(len(f)))
    remove_bom = f.replace('\ufeff', '###')
    comma_delimit = remove_bom.replace('\n', ',').strip().lower().replace('\r',␣
 ↪'').split(',')
    clean_text = [each for each in comma_delimit if (str.rstrip(each) != '') or\
                  (str.rstrip(each) not in string_punctuation)]
    return pd.DataFrame({'text': clean_text})

def clean(s):
    '''
    This function takes in a string and remove punctuation, numeric values
    and all extra spaces from string.
    '''
    string_punctuation = '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~‘êàé-’”“'
    # remove punctuation
    no_punc = s.translate(str.maketrans('', '', string_punctuation))
    # remove num
    no_num = ''.join([each for each in no_punc if not each.isdigit()])
    # remove extra spaces
    return ' '.join(no_num.split())


def count_alpha(x):
    '''
    This function takes in a string and returns the occurances of each letter
    in the string.
    '''
    return Counter(x)

def count(s):
    '''
    This function takes in a string and manually counts the occurances
    of each letter in  the string.
    '''
    count_a = s.count('a')
    count_b = s.count('b')
    count_c = s.count('c')
    count_d = s.count('d')
    count_e = s.count('e')
    count_f = s.count('f')
    count_g = s.count('g')
    count_h = s.count('h')
    count_i = s.count('i')
    count_j = s.count('j')
```

```
    count_k = s.count('k')
    count_l = s.count('l')
    count_m = s.count('m')
    count_n = s.count('n')
    count_o = s.count('o')
    count_p = s.count('p')
    count_q = s.count('q')
    count_r = s.count('r')
    count_s = s.count('s')
    count_t = s.count('t')
    count_u = s.count('u')
    count_v = s.count('v')
    count_w = s.count('w')
    count_x = s.count('x')
    count_y = s.count('y')
    count_z = s.count('z')
    count_space = s.count(' ')

    return {'a': count_a, 'b': count_b, 'c': count_c,
            'd': count_d, 'e': count_e, 'f': count_f,
            'g': count_g, 'h': count_h, 'i': count_i,
            'j': count_j, 'k': count_k, 'l': count_l,
            'm': count_m, 'n': count_n, 'o': count_o,
            'p': count_p, 'q': count_q, 'r': count_r,
            's': count_s, 't': count_t, 'u': count_u,
            'v': count_v, 'w': count_w, 'x': count_x,
            'y': count_y, 'z': count_z, 'space': count_space}

def markov(s):
    '''
    This is a helper function that takes in dict and returns
    the value of the argument.
    '''
    return markov_pred_dict[s]

def markov_sampler(char_init, n_iter, markov_dict):
    '''
    This function takes in a series of initial parameters
    and recursively generates a string based on the most
    common letter transistions from the text.
    '''
    char_now = char_init
    markov_str = []
    n_iter = n_iter
    for i in range(n_iter):
        char_now = markov(char_now)
        markov_str.append(char_now)
```

```
    return ''.join(markov_str)

def generate_kgram(s, n):
    '''
    This function takes in a string and counts all of the possible
    k-grams found in  the string.
    '''
    return Counter([s[i:i+n] for i in range(0, len(s), 1)])

def graph(x, y, data, ylabel, xlabel, title):
    '''
    This is a helper function that is used to create the bar charts
    needed to show the distribution of letters in  the text.
    '''
    sns.set(rc={'figure.figsize':(15,5)})
    sns.barplot(x=x, y=y, data=data)
    plt.ylabel(ylabel, fontsize=16)
    plt.xlabel(xlabel, fontsize=16)
    plt.ylim(0, 0.180, 0.025)
    plt.xticks(fontsize=15)
    plt.yticks(fontsize=15)
    plt.title(title, fontsize=17)
    return plt.show()

idx_list = ['space', 'a', 'b', 'c',
            'd', 'e', 'f', 'g', 'h',
            'i', 'j', 'k', 'l', 'm',
            'n', 'o', 'p', 'q', 'r',
            's', 't', 'u', 'v', 'w',
            'x', 'y', 'z']

new_col_list = ['first_pos', ' ', 'a', 'b', 'c',
                'd', 'e', 'f', 'g', 'h', 'i',
                'j', 'k', 'l', 'm', 'n', 'o',
                'p', 'q', 'r', 's', 't', 'u',
                'v', 'w', 'x', 'y', 'z']

idx_list_2 = ['first_pos', 'a', 'b', 'c',
            'd', 'e', 'f', 'g', 'h',
            'i', 'j', 'k', 'l', 'm',
            'n', 'o', 'p', 'q', 'r',
            's', 't', 'u', 'v', 'w',
            'x', 'y', 'z']

df = load_data(url)
df.shape
```

```
Text File Has Been Downloaded
---
This Text File Contains: 790332 Characters
```

[2]:
```
(20804, 1)
```

[3]:
```
'''
This code applies the user-defined cleaning function and
adds the clean text to a  new Pandas column. The function
also inserts a np.nan each time that there is a null value.
'''
df_clean = copy.deepcopy(df)
df_clean.loc[:, 'clean_string'] = df_clean['text'].apply(clean)
df_clean.replace('', np.nan, inplace=True)
df_clean.describe()
```

[3]:

	text	clean_string
count	20804	20782
unique	18816	18197
top	and	and
freq	177	191

[4]:
```
'''
This code returns all of the rows that does not have a null.
and returns the summary statistics.
'''
df_clean = df_clean[~df_clean['clean_string'].isna()]
df_clean.describe()
```

[4]:

	text	clean_string
count	20782	20782
unique	18810	18197
top	and	and
freq	177	191

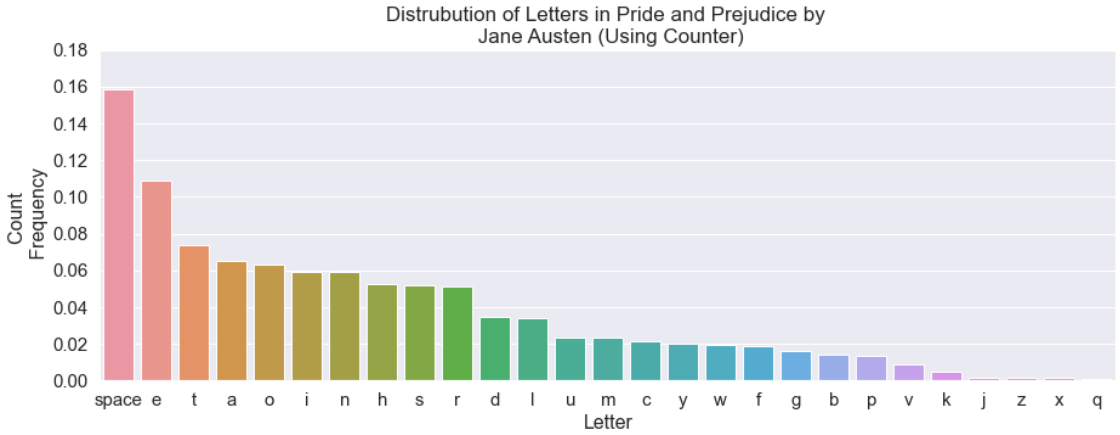
[5]:
```
'''
This code creates a dict of all of the occurances of the string and
inserts the results into a new column using two methods. The first method
manually counts the occurances and the second uses the built in Python method
to count the occurances.
'''
df_clean.loc[:, 'clean_string_count'] = df_clean['clean_string'].
 ↪apply(count_alpha)
df_clean.loc[:, 'clean_string_count_py'] = df_clean['clean_string'].apply(count)

'''
This code summarizes all of the counts from the list of dict and returns a
```

```
final dict that has the final counts of letters in the body of text.
'''
list_dict = [dict(each) for each in df_clean.clean_string_count.tolist()]
final_dist = {}
for d in list_dict:
    for k in d.keys():
        final_dist[k] = final_dist.get(k, 0) + d[k]

list_dict_py = [dict(each) for each in df_clean.clean_string_count_py.tolist()]
final_dist_py = {}
for d in list_dict_py:
    for k in d.keys():
        final_dist_py[k] = final_dist_py.get(k, 0) + d[k]
```

[6]:
```
'''
Create df_freq DataFrame of letter occurances from the manual counts.

This code fulfills the requirements outlined in 2(b) of the homework␣
 ↪instructions.
'''
df_freq = pd.DataFrame(final_dist.items(), columns=['letter', 'count'])
df_freq['freq'] = df_freq['count'] / df_freq['count'].sum()
df_freq.at[3, 'letter'] = 'space'
df_freq.sort_values(by='count', ascending=False, inplace=True)
df_freq.reset_index(drop=True, inplace=True)
df_freq['rank'] = df_freq.index + 1


'''
Create df_freq_py DataFrame of letter occurances from the built-in library.
'''
df_freq_py = pd.DataFrame(final_dist_py.items(), columns=['letter', 'count'])
df_freq_py['freq'] = df_freq_py['count'] / df_freq_py['count'].sum()
#df_freq_py.at[3, 'letter'] = 'space'
df_freq_py.sort_values(by='count', ascending=False, inplace=True)
df_freq_py.reset_index(drop=True, inplace=True)
df_freq_py['rank'] = df_freq_py.index + 1

'''
Check if both df are the same. A True suggests that both
methods produce the same results and that the results of the
manual counts match those produced by the built in counter.
'''
df_freq_py.equals(df_freq)
```

[6]:
```
True
```

[7]:
```
'''
Uses the user defined function to print a histogram of the letter counts.
'''
graph(x='letter',
      y='freq',
      data=df_freq,
      ylabel='Count \n Frequency',
      xlabel='Letter',
      title='Distrubution of Letters in Pride and Prejudice by \n Jane Austen␣
 ↪(Using Counter)')
```



[8]:
```
'''
Applies the user-defined function to produce the k-grams
from the text and stores the results as a dict in a new column.
'''
df_clean.loc[:, 'kgrams'] = df_clean['clean_string'].apply(generate_kgram,␣
 ↪args=[2])
df_clean.head()
```

[8]:

```
                                                text  \
0  the project gutenberg ebook of pride and preju...
1                                       by jane austen
2  this ebook is for the use of anyone anywhere a...
3  almost no restrictions whatsoever.  you may co...
4                                     give it away or

                                          clean_string  \
0  the project gutenberg ebook of pride and preju...
1                                       by jane austen
2  this ebook is for the use of anyone anywhere a...
3  almost no restrictions whatsoever you may copy it
4                                     give it away or
```

```
                                clean_string_count  \
0  {'t': 3, 'h': 1, 'e': 8, ' ': 7, 'p': 3, 'r': ...
1  {'b': 1, 'y': 1, ' ': 2, 'j': 1, 'a': 2, 'n': ...
2  {'t': 5, 'h': 4, 'i': 3, 's': 4, ' ': 13, 'e':...
3  {'a': 3, 'l': 1, 'm': 2, 'o': 6, 's': 4, 't': ...
4  {'g': 1, 'i': 2, 'v': 1, 'e': 1, ' ': 3, 't': ...

                             clean_string_count_py  \
0  {'a': 1, 'b': 2, 'c': 2, 'd': 3, 'e': 8, 'f': ...
1  {'a': 2, 'b': 1, 'c': 0, 'd': 0, 'e': 2, 'f': ...
2  {'a': 4, 'b': 1, 'c': 1, 'd': 1, 'e': 6, 'f': ...
3  {'a': 3, 'b': 0, 'c': 2, 'd': 0, 'e': 3, 'f': ...
4  {'a': 2, 'b': 0, 'c': 0, 'd': 0, 'e': 1, 'f': ...

                                             kgrams
0  {'th': 1, 'he': 1, 'e ': 2, ' p': 3, 'pr': 3, ...
1  {'by': 1, 'y ': 1, ' j': 1, 'ja': 1, 'an': 1, ...
2  {'th': 3, 'hi': 1, 'is': 2, 's ': 2, ' e': 1, ...
3  {'al': 1, 'lm': 1, 'mo': 1, 'os': 1, 'st': 2, ...
4  {'gi': 1, 'iv': 1, 've': 1, 'e ': 1, ' i': 1, ...
```

[9]:
```
'''
This code summarizes all of the counts from the list of dict and returns a
final dict that has the final counts of letters in the body of text.
'''
kgram_list_dict_py = [dict(each) for each in df_clean.kgrams.tolist()]
kgram_dist= {}
for d in kgram_list_dict_py:
    for k in d.keys():
        kgram_dist[k] = kgram_dist.get(k, 0) + d[k]

'''
This code creates a DataFrame containing the kgrams and splits them up
so that we can see the first and second letters of the k-gram as well as
the counts.
'''
df_kgram = pd.DataFrame(kgram_dist.items(), columns=['kgram', 'count'])
df_kgram['kgram_len'] = df_kgram['kgram'].str.split(' ').str.len()
two_grams = df_kgram[df_kgram.kgram_len == 1]

two_grams = copy.deepcopy(two_grams)

two_grams.loc[:, 'first_pos'] = two_grams['kgram'].str[0]
two_grams.loc[:, 'second_pos'] = two_grams['kgram'].str[1]

df = two_grams[['kgram', 'count', 'first_pos', 'second_pos']]
```

```
df.head()
```

[9]:
```
  kgram  count first_pos second_pos
0    th  14098         t          h
1    he  15044         h          e
4    pr   1494         p          r
5    ro   2060         r          o
6    oj     89         o          j
```

[10]:
```
'''
This code creates a square adjacency matrix of the letter occurances found in␣
 ↪the text.
'''
df_trans = df.pivot_table(index=['first_pos'], columns='second_pos',␣
 ↪values='count')
df_trans['first_pos'] = df_trans.index
df_trans.reset_index(drop=True, inplace=True)
df_reorder = df_trans.reindex(columns=idx_list_2)
df_reorder.head()
```

[10]:
```
second_pos first_pos       a       b       c       d       e      f      g  \
0                  a     7.0  1562.0  1160.0  2447.0     8.0  473.0  899.0
1                  b   292.0     7.0     NaN     1.0  4290.0    NaN    NaN
2                  c  1441.0     1.0   344.0     3.0  2575.0    NaN    NaN
3                  d  1114.0     8.0     1.0   215.0  2623.0   25.0  124.0
4                  e  3165.0    78.0  1802.0  5544.0  1960.0  709.0  333.0

second_pos       h       i  ...      q       r       s       t       u  \
0              6.0  1610.0  ...    NaN  4270.0  4903.0  5818.0   517.0
1              7.0   435.0  ...    NaN   325.0   156.0   111.0  1144.0
2           2371.0   551.0  ...  100.0   331.0     7.0  1440.0   375.0
3             12.0  2019.0  ...    1.0   244.0   428.0    21.0   240.0
4            154.0  1087.0  ...  198.0  11652.0  3739.0  2624.0    23.0

second_pos       v      w      x       y     z
0           1371.0  348.0   11.0  1229.0  23.0
1              1.0    NaN    NaN   692.0   NaN
2              NaN    NaN    NaN   509.0   NaN
3            100.0   22.0    NaN   407.0   NaN
4           1430.0  389.0  745.0  1171.0   NaN

[5 rows x 27 columns]
```

[11]:
```
'''
This code find the maximum value in each row (letter) and returns the␣
 ↪corresponding
```

```
letter that represents the highest co-occuring letter in the given the first␣
 ↪letter.
As an example, the entry "a": "n" means that for the letter 'a', the most common␣
 ↪letter
that follows is 'n'. These data only includes the 26 letters of the alphabet (no␣
 ↪space).

This code fulfills the requirements outlined in 2(c) of the homework␣
 ↪instructions.
'''

df_reorder['idxmax'] = df_reorder.iloc[:, 1:-1].idxmax(axis=1)
markov_pred_dict = dict(zip(df_reorder['first_pos'].tolist(),␣
 ↪df_reorder['idxmax'].tolist()))
markov_pred_dict
```

[11]:
```
{'a': 'n',
 'b': 'e',
 'c': 'o',
 'd': 'e',
 'e': 'r',
 'f': 'o',
 'g': 'h',
 'h': 'e',
 'i': 'n',
 'j': 'e',
 'k': 'e',
 'l': 'e',
 'm': 'e',
 'n': 'd',
 'o': 'u',
 'p': 'e',
 'q': 'u',
 'r': 'e',
 's': 'e',
 't': 'h',
 'u': 'r',
 'v': 'e',
 'w': 'a',
 'x': 'p',
 'y': 'o',
 'z': 'a'}
```

[12]:
```
'''
Uses the dict showing the most common transistion letter, this code produces a␣
 ↪3,000 character
```

```
string starting with the letter 't'.

This code fulfills the requirements outlined in 2(e) of the homework␣
↪instructions.
'''
markov_string = markov_sampler(char_init='t', n_iter=3000,␣
↪markov_dict=markov_pred_dict)
markov_string
```

```
[12]: 'hererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
erererererererererererererererererererererererererererererererererererererererer
ererererererererererererererererererererere'
```

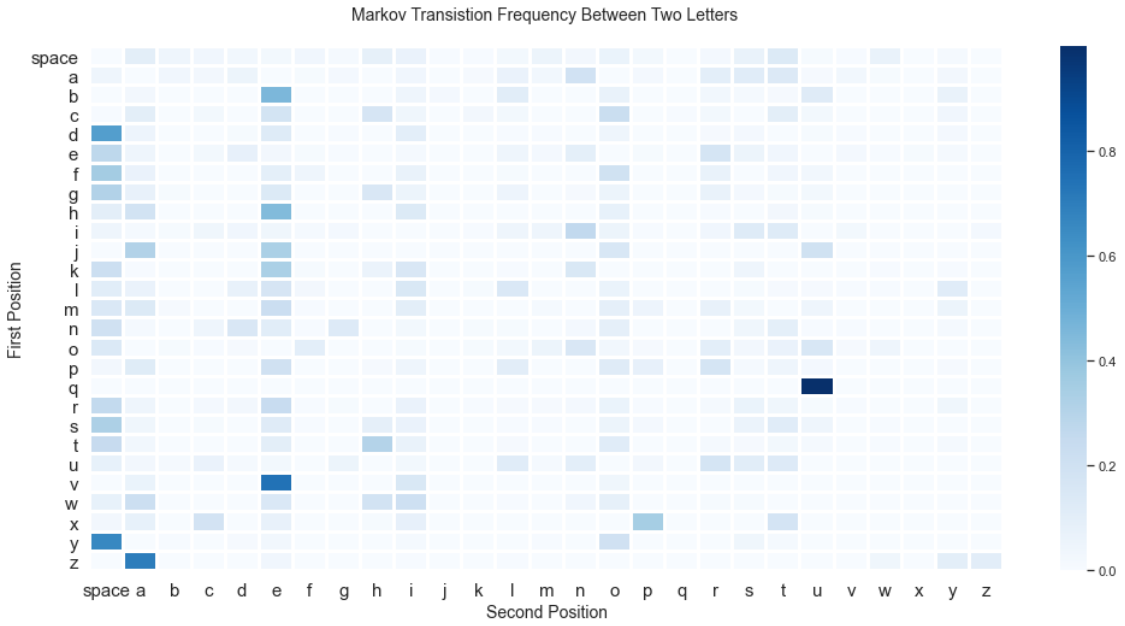
[13]:
```
'''
This code creates a square adjacency matrix of the letter occurances found
in the text and includes the space.
'''
df_kgram = copy.deepcopy(df_kgram)
df_kgram.loc[:, 'first_pos'] = df_kgram['kgram'].str[0]
df_kgram.loc[:, 'second_pos'] = df_kgram['kgram'].str[1]
df_k = df_kgram[['kgram', 'count', 'first_pos', 'second_pos']]
df_ktrans = df_k.pivot_table(index=['first_pos'], columns='second_pos',␣
 ↪values='count')
df_ktrans['first_pos'] = df_ktrans.index
df_ktrans.reset_index(drop=True, inplace=True)
df_ktrans.fillna(0, inplace=True)

df_ktrans = df_ktrans.reindex(columns=new_col_list).fillna(0)
df_ktrans.rename(columns={' ': 'space'}, inplace=True)
df_ktrans.loc[0, 'first_pos'] = 'space'
df_ktrans['total'] = df_ktrans.iloc[:, 0:].sum(axis=1)
df_ktrans.head()
```

[13]:
```
second_pos first_pos    space        a       b       c       d       e  \
0              space      0.0  10684.0  5187.0  4012.0  3368.0  2701.0
1                  a   1981.0      7.0  1562.0  1160.0  2447.0     8.0
2                  b     13.0    292.0     7.0     0.0     1.0  4290.0
3                  c     78.0   1441.0     1.0   344.0     3.0  2575.0
4                  d  11770.0   1114.0     8.0     1.0   215.0  2623.0

second_pos       f       g       h  ...       r       s        t       u  \
0           3645.0  1675.0  9233.0  ...  2049.0  7289.0  13894.0   894.0
1            473.0   899.0     6.0  ...  4270.0  4903.0   5818.0   517.0
2              0.0     0.0     7.0  ...   325.0   156.0    111.0  1144.0
3              0.0     0.0  2371.0  ...   331.0     7.0   1440.0   375.0
4             25.0   124.0    12.0  ...   244.0   428.0     21.0   240.0

second_pos       v       w     x       y     z     total
0            791.0  7469.0   0.0  1915.0   1.0  103750.0
1           1371.0   348.0  11.0  1229.0  23.0   42456.0
2              1.0     0.0   0.0   692.0   0.0    9363.0
3              0.0     0.0   0.0   509.0   0.0   14101.0
4            100.0    22.0   0.0   407.0   0.0   20646.0

[5 rows x 29 columns]
```

[14]:
```
'''
This code creates a square adjacency matrix of the letter occurances found
in the text and includes the space.
'''
```

```
df_ktrans_freq = df_ktrans.iloc[:, 1:].div(df_ktrans['total'] , axis=0)
df_ktrans_freq_2 = df_ktrans_freq.iloc[:, 0:27]
df_ktrans_freq_2['idx'] = idx_list
df_ktrans_freq_2.set_index('idx', inplace=True)
```

[15]:
```
'''
Creates a heatmap using the data in the adjacency matrix that shows
the most common letter transistion in the text.

This code fulfills the requirements outlined in 2(d) of the homework␣
 ↪instructions.
'''
sns.set(rc={'figure.figsize':(17,8)})
sns.heatmap(df_ktrans_freq_2.iloc[:, 0:27], linewidths=2, yticklabels=1,␣
 ↪cmap='Blues')
plt.ylabel('First Position', fontsize=14)
plt.xlabel('Second Position', fontsize=14)
plt.yticks(fontsize=15)
plt.xticks(fontsize=15)
plt.title('Markov Transistion Frequency Between Two Letters \n', fontsize=14)
plt.show()
```



Letter in Second Position

Letter in First Position	space	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
space	0	10684	5187	4012	3368	2701	3645	1675	9233	6890	487	609	2684	5743	3059	6495	2766	210	2049	7289	13894	894	791	7469	0	1915	1
a	1981	7	1562	1160	2447	8	473	899	6	1610	0	591	2879	1339	8132	16	856	0	4270	4903	5818	517	1371	348	11	1229	23
b	13	292	7	0	1	4290	0	0	7	435	185	0	1029	4	1	670	0	0	325	156	111	1144	1	0	0	692	0
c	78	1441	1	344	3	2575	0	0	2371	551	0	431	349	0	0	3195	0	100	331	7	1440	375	0	0	0	509	0
d	11770	1114	8	1	215	2623	25	124	12	2019	1	1	151	86	69	964	0	1	244	428	21	240	100	22	0	407	0
e	18631	3165	78	1802	5544	1960	709	333	154	1087	50	86	3312	1436	6349	98	653	198	11652	3739	2624	23	1430	389	745	1171	0
f	4222	789	1	6	6	1145	588	2	8	847	0	0	136	0	2	2336	1	0	865	7	425	380	0	2	0	45	0
g	3101	735	90	1	5	1322	2	35	1526	550	0	0	502	9	121	556	6	0	707	229	99	241	0	1	0	24	0
h	3499	6413	53	0	9	15044	11	0	4	4572	0	0	43	84	6	2662	0	0	166	75	947	275	0	4	0	78	0
i	1946	735	338	1561	1368	1517	756	895	5	2	0	236	1779	1730	10291	2254	182	5	1481	4987	4988	22	927	1	56	0	770
j	0	303	0	0	0	322	0	0	0	1	0	0	0	0	0	152	0	0	0	0	0	191	0	0	0	0	0
k	702	17	1	1	0	1058	37	2	195	489	0	0	18	0	475	2	0	0	1	145	0	0	0	15	0	17	0
l	2359	1490	5	32	1679	3652	629	19	4	3218	5	240	3163	77	27	1347	48	1	31	227	261	336	108	125	0	2493	0
m	2134	1960	203	0	0	3237	58	0	1	1516	1	0	16	241	23	1319	738	0	1129	360	8	729	0	1	0	829	0
n	7516	601	99	1689	5730	4061	236	4878	38	914	69	365	408	14	786	3401	22	73	14	1576	3525	170	252	34	47	636	1
o	5801	108	349	239	559	98	4081	108	101	339	89	394	1055	2262	6263	1326	630	4	4102	1141	2737	6369	542	1853	6	85	2
p	255	1023	3	0	2	1748	1	10	123	430	0	0	932	5	0	1055	714	0	1494	164	402	185	0	3	0	129	0
q	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	637	0	0	0	0	0
r	8197	1431	58	701	862	7560	218	318	113	2127	1	204	464	467	619	2060	133	0	589	2043	1339	297	216	88	0	1337	0
s	10421	1356	59	343	36	3915	129	35	2953	2212	0	99	141	134	21	1841	695	4	7	1926	3588	1527	2	130	0	65	0
t	11489	1600	7	100	3	4719	91	1	14098	3279	0	0	703	128	76	5276	11	0	879	580	1206	770	0	241	1	786	37
u	1237	461	245	1039	273	394	52	865	2	366	0	1	1784	249	1503	7	467	0	2640	1728	1995	0	1	0	1	5	3
v	0	368	0	0	0	4334	0	0	0	886	0	0	0	0	0	245	0	0	1	0	0	3	0	0	0	8	0
w	995	2738	2	7	6	1810	3	5	2345	2581	0	13	63	1	468	1075	1	0	131	99	2	0	0	12	0	0	0
x	26	69	0	160	0	67	1	0	7	73	0	0	0	0	0	0	300	1	0	0	151	3	0	0	0	1	0
y	7376	14	66	2	172	358	9	0	3	180	0	0	18	50	17	2231	7	0	18	469	189	0	0	8	0	2	0
z	0	659	0	0	0	34	0	0	0	6	0	0	4	0	0	0	0	0	0	0	0	0	0	37	0	97	101

Note:
Numbers in the tables represent counts of the bigrams (digraphs).