A_Adams_aja149_hw1

March 1, 2022

1 Alexander Adams

2 aja149

3 Homework #1: Words, Words, Words

4 PPOL628 Text as Data

5 Homework: State Your Assumptions

> POLONIUS What do you read, my lord?
>
> HAMLET Words, words, words
>
> – Hamlet, Act 2, Scene 2

5.1 This homework deals with the assumptions made when taking text from its original "raw" form into something more computable.

- Assumptions about the shape of text (e.g. how to break a corpus into documents)
- Assumptions about what makes a token, an entity, etc.
- Assumptions about what interesting or important content looks like, and how that informs our analyses.

5.2 There are three parts:

1. Splitting Lines from Shakespeare
2. Tokenizing and Aligning lines into plays
3. Assessing and comparing characters from within each play

NB This file is merely a template, with instructions; do not feel constrained to using it directly if you do not wish to.

5.3 Get the Data

Since the class uses dvc, it is possible to get this dataset either using the command line (e.g. dvc import https://github.com/TLP-COI/text-data-course re-

sources/data/shakespeare/shakespeare.txt), or using the python api (if you wish to use python):

[1]:
```
#I could not get the duc.api read function to work. It consistenly produced the␣
 ↪following error:
#AttributeError: 'HashFile' object has no attribute 'get'

#I switched to this import method because it was the only way I could actually␣
 ↪read in the text file.

#from duc.api import read,get_url
#import pandas as pd

#txt = pd.read_csv('shakespeare.txt', sep = '\n', header = None)

#print(txt[:250])
```

[2]:
```
#Use relative path because shakespeare.txt is in parent directory
with open('..\..\..\shakespeare.txt') as f:
    lines = f.read()
```

Make sure this works before you continue! Either way, it would likely be beneficial to have the data downloaded locally to keep from needing to re-dowload it every time.

---

6 Part 1

Split the text file into a table, such that:

```
-each row is a single line of dialogue
-there are columns for:
    *the speaker
    *the line number
    *the line dialogue (the text)
```

Hint: you will need to use RegEx to do this rapidly. See the in-class "markdown" example!

Question(s):

```
*What assumptions have you made about the text that allowed you to do this?
```

---

[3]:
```
#Import the re package for regex
import pandas as pd
pd.set_option('display.max_rows', None)
pd.set_option('display.width', None)
pd.set_option('display.max_colwidth', None)
import re
```

Each chunk of text contains two parts, a speaker and the dialogue.

The key assumptions to be made here are that the speaker's name is always followed by a colon and a new line(":"), and each chunk of text is followed by two new lines:

Speaker_Name:Text

Note: In the interest of full disclosure, I initially tried to get away with using as little regex for this part as possible. I've preserved my original solution (which works perfectly fine as far as I can tell, but eschews regex almost entirely in favor of pandas string methods) as commented code below.

The regex patterns below were posted by Professor Sexton and Ella Zhang in our class slack. I've included original explanations, and tried to modify the patterns so that I wasn't just copying outright. (The only real modification I made was swapping out a plus sign in the first pattern for {1,}. While this is functionally the same but longer and therefore worse, it's at least an original contribution to this.)

[103]:
```
#Goal: create two patterns
#1) Pattern to capture speaker
#2) Pattern to capture dialogue

shakespeare_pattern = re.compile(
    "(^[A-Z].{1,}?):$" #Speaker pattern
    "\n{1}(.*?)\n{2}", #Dialogue pattern
    flags = re.S | re.M
)
```

The first pattern above is as follows: (̂[A-Z].+?):$. First, let's consider the characters inside the parentheses. The caret indicates the start of a pattern. The square brackets with A-Z in them mean that there will be a capital letter. Periods can be any character, and a character followed by a number in curly braces must occur that many times. In this case, the number is followed by a comma, which means the preceding token can occur that number of times or more. The question mark modifies the number-comma in curly braces by making it "lazy". In regex terms, this means that while it can match any number of characters corresponding to the preceding token, a lazy version will match the minimum characters necessary. Taken all together, this pattern means that the line must start with a capital letter, then be followed by any sequence of characters, as long as it takes but as few as possible to fulfill the pattern. The colon is standard character (in this case, it's intended to match the colon after the speaker's name), and the dollar sign ends the line.

The second pattern above is as follows: 1(.*?)2. Conceptually, this one is a bit simpler. "" indicates a new line, and a number in curly brackets hard-codes the number of times the preceding token can occur. Thus, 1 will match an occurrence of one new line. The parentheses demarcate a group, as before. The period can be anything, and the asterisk means that token can occur any number of times. The question mark, as before, makes the asterisk lazy. All together, this parenthetical group matches any token, any number of times, as few tokens as possible. This pattern is then followed by 2, which matches two new lines. All together, this pattern captures the new line at the start of the line, then all the textual characters on the row, and then the two lines which separate this chunk of text from the next one.

The last parts of the regex expression above are the two flags. In the `re` package for Python, `re.S` allows the period to match a new line character. Without this flag, the period can match any

character except a new line. The `re.M` flag lets the caret and dollar sign work at the start and end of a given line, rather than just at the start and end of a whole string. These flags are separated by an OR operator, indicating that only one needs to match for a match to be valid.

[106]:
```
text = shakespeare_pattern.findall(lines)
```

[109]:
```
text = pd.DataFrame(matches1)
```

[110]:
```
#Rename columns for ease of use
text = text.rename(columns = {0: 'Speaker',
                              1: 'Text'})
```

[4]:
```
#Original solution with minimal regex:

#1) Split the text using two new lines as a delimiter
#text = re.split(r'\n\n',lines)

#2) Convert to a pandas dataframe
#text = pd.DataFrame(text, columns = ['text'])

#3) Call the split method on the string attribute to split the text column into␣
 ↪2,
#using the colon followed by a new line as the delimiter.
#text = text['text'].str.split(':\n', 1, expand=True)

#~~4) Then use the `split` method again to break each chunk up into individual␣
 ↪lines, followed by
#the `explode` function to make each individual line its own row. Call␣
 ↪`reset_index` with
#`drop = True` to drop the index column, which has the same value for all lines␣
 ↪which were part
#of the same original chunk.~~

#EDIT: As per the discussion in the slack channel, this step is not necessary.␣
 ↪I've included
#the below two cells, commented out, to show what I would have done to split␣
 ↪each chunk of
#dialogue into individual lines.

#text[1] = text[1].str.split('\n')
#text = text.explode(1).reset_index(drop = True)


#5) Finally, reset the index to create a column of integer values for line␣
 ↪numbers,
#and then add one to each item in that column because Python is a zero-indexed␣
 ↪language.
```

```
#Rename the columns in the data frame for clarity.
#text = text.reset_index().rename(columns = {'index': 'Line_Number',
#                                            0: 'Speaker',
#                                            1: 'Text'})
#text['Line_Number'] = text['Line_Number']+1
#text.head(10)
```

7 Part 2

You have likely noticed that the lines are not all from the same play! Now, we will add some useful metadata to our table:

- Determine a likely source title for each line.
- Add the title as a 'play' column in the data table.
- Make sure to document your decisions, assumptions, external data sources, etc.

This is fairly open-ended, and you are not being judged completely on accuracy. Instead, think outside the box a bit as to how you might accomplish this, and attempt to justify whatever approximations or assumptions you felt were appropriate.

[111]:
```
#Just for fun, let's see a list of all unique speakers in the table:
text.Speaker.unique()
```

[111]:
```
array(['First Citizen', 'All', 'Second Citizen', 'MENENIUS', 'MARCIUS',
       'Messenger', 'First Senator', 'COMINIUS', 'TITUS', 'SICINIUS',
       'BRUTUS', 'AUFIDIUS', 'Second Senator', 'VOLUMNIA', 'VIRGILIA',
       'Gentlewoman', 'VALERIA', 'LARTIUS', 'First Soldier',
       'Second Soldier', 'First Roman', 'Second Roman', 'Third Roman',
       'Lieutenant', 'CORIOLANUS', 'Both', 'Herald', 'First Officer',
       'Second Officer', 'Officer', 'Senators', 'Third Citizen',
       'Fourth Citizen', 'Fifth Citizen', 'Both Citizens',
       'Sixth Citizen', 'Seventh Citizen', 'All Citizens', 'Citizens',
       'Senators, &C', 'AEdile', 'A Patrician', 'Second Patrician',
       'Both Tribunes', 'Roman', 'Volsce', 'Citizen', 'First Servingman',
       'Second Servingman', 'Third Servingman', 'Second Messenger',
       'Young MARCIUS', 'First Conspirator', 'Second Conspirator',
       'Third Conspirator', 'All The Lords', 'Lords', 'First Lord',
       'All Conspirators', 'All The People', 'Second Lord', 'Third Lord',
       'GLOUCESTER', 'CLARENCE', 'BRAKENBURY', 'HASTINGS', 'LADY ANNE',
       'Gentleman', 'GENTLEMEN', 'RIVERS', 'GREY', 'QUEEN ELIZABETH',
       'BUCKINGHAM', 'DERBY', 'QUEEN MARGARET', 'DORSET', 'CATESBY',
       'First Murderer', 'Second Murderer', 'Second murderer',
       'KING EDWARD IV', 'Boy', 'DUCHESS OF YORK', 'Girl', 'Children',
       'ARCHBISHOP OF YORK', 'YORK', 'PRINCE EDWARD', 'Lord Mayor',
       'CARDINAL', 'STANLEY', 'LORD STANLEY', 'Pursuivant', 'Priest',
       'RATCLIFF', 'VAUGHAN', 'BISHOP OF ELY', 'LOVEL', 'Scrivener',
       'ANOTHER', 'KING RICHARD III', 'Page', 'TYRREL', 'Third Messenger',
```

```
'Fourth Messenger', 'CHRISTOPHER', 'Sheriff', 'RICHMOND', 'OXFORD',
'HERBERT', 'BLUNT', 'SURREY', 'NORFOLK', 'Ghost of Prince Edward',
'Ghost of CLARENCE', 'Ghost of GREY', 'Ghosts of young Princes',
'Ghost of BUCKINGHAM', 'LORDS', 'KING RICHARD II', 'JOHN OF GAUNT',
'HENRY BOLINGBROKE', 'THOMAS MOWBRAY', 'DUCHESS', 'Lord Marshal',
'DUKE OF AUMERLE', 'First Herald', 'Second Herald', 'GREEN',
'BUSHY', 'DUKE OF YORK', 'QUEEN', 'NORTHUMBERLAND', 'LORD ROSS',
'LORD WILLOUGHBY', 'Servant', 'BAGOT', 'HENRY PERCY',
'LORD BERKELEY', 'Captain', 'EARL OF SALISBURY',
'BISHOP OF CARLISLE', 'SIR STEPHEN SCROOP', 'Lady', 'Gardener',
'GARDENER', 'LORD FITZWATER', 'Lord', 'DUKE OF SURREY', 'Abbot',
'EXTON', 'Groom', 'Keeper', 'SAMPSON', 'GREGORY', 'ABRAHAM',
'BENVOLIO', 'TYBALT', 'CAPULET', 'LADY CAPULET', 'MONTAGUE',
'LADY MONTAGUE', 'PRINCE', 'ROMEO', 'PARIS', 'Nurse', 'JULIET',
'MERCUTIO', 'First Servant', 'Second Servant', 'Second Capulet',
'Chorus', 'FRIAR LAURENCE', 'PETER', 'NURSE', 'LADY  CAPULET',
'First Musician', 'Second Musician', 'Musician', 'Third Musician',
'BALTHASAR', 'Apothecary', 'FRIAR JOHN', 'PAGE', 'First Watchman',
'Second Watchman', 'Third Watchman', 'WARWICK', 'EDWARD',
'RICHARD', 'KING HENRY VI', 'CLIFFORD', 'WESTMORELAND', 'EXETER',
'JOHN MORTIMER', 'RUTLAND', 'Tutor', 'GEORGE', 'Son', 'Father',
'First Keeper', 'Second Keeper', 'LADY GREY', 'Nobleman',
'KING LEWIS XI', 'BONA', 'Post', 'SOMERSET', 'Huntsman', 'Mayor',
'Soldier', 'First Messenger', 'ARCHIDAMUS', 'CAMILLO', 'POLIXENES',
'LEONTES', 'HERMIONE', 'MAMILLIUS', 'First Lady', 'Second Lady',
'ANTIGONUS', 'Gaoler', 'PAULINA', 'EMILIA', 'CLEOMENES', 'DION',
'Mariner', 'Shepherd', 'Clown', 'Time', 'AUTOLYCUS', 'FLORIZEL',
'PERDITA', 'DORCAS', 'MOPSA', 'Shepard', 'First Gentleman',
'Second Gentleman', 'Third Gentleman', 'DUKE VINCENTIO', 'ESCALUS',
'ANGELO', 'DUKE', 'LUCIO', 'MISTRESS OVERDONE', 'POMPEY',
'CLAUDIO', 'Provost', 'FRIAR THOMAS', 'ISABELLA', 'FRANCISCA',
'ELBOW', 'FROTH', 'POMPHEY', 'Justice', 'MARIANA', 'ABHORSON',
'BARNARDINE', 'FRIAR PETER', 'SLY', 'Hostess', 'First Huntsman',
'Second Huntsman', 'Players', 'A Player', 'Third Servant', 'ALL',
'LUCENTIO', 'TRANIO', 'BAPTISTA', 'GREMIO', 'HORTENSIO',
'KATHARINA', 'HORTENSIA', 'BIANCA', 'BIONDELLO', 'PETRUCHIO',
'GRUMIO', 'KATARINA', 'CURTIS', 'NATHANIEL', 'PHILIP', 'JOSEPH',
'NICHOLAS', 'ALL SERVING-MEN', 'Pedant', 'Tailor', 'VINCENTIO',
'Widow', 'Master', 'Boatswain', 'ALONSO', 'ANTONIO', 'GONZALO',
'SEBASTIAN', 'Mariners', 'MIRANDA', 'PROSPERO', 'ARIEL', 'CALIBAN',
'FERDINAND', 'ADRIAN', 'FRANCISCO'], dtype=object)
```

Some of the names are in all caps, while some are in sentence case. I'll adjust so they're all in sentence case.

[112]:
```
text['Speaker'] = text['Speaker'].str.title()
```

Through Googling, I found a table of all characters in all Shakepeare plays, along with the number

of lines spoken by that character and the play they appear in. When I tried to scrape the table using the `read_html` function from `pandas`, I received a 403 error code, indicating forbidden access. Fortunately, I was able to take a much cruder approach and copy/paste the contents of the table into an Excel file, which I've committed to my branch/folder for this homework assignment. This dataset comes from playshakespeare.com, specifically this page: https://www.playshakespeare.com/study/complete-shakespeare-character-list. All credit goes to the original compilers of the data.

[12]:
```
#Unsuccessful scraping attempt:
#characters = pd.read_html('https://www.playshakespeare.com/study/
 ↪complete-shakespeare-character-list')
```

[13]:
```
characters = pd.read_csv('shakespeare_characters.csv')
```

[14]:
```
characters.head(10)
```

[14]:
```
                                cter Lines                        Play
0               King of France (KING.)   373  All's Well That Ends Well
1             Duke of Florence (DUKE.)    19  All's Well That Ends Well
2  Bertram, Count of Roussillion (BER.)   240  All's Well That Ends Well
3                         Lafew (LAF.)   146  All's Well That Ends Well
4                      Parolles (PAR.)   184  All's Well That Ends Well
5                      Rinaldo (STEW.)    23  All's Well That Ends Well
6                       Lavatch (CLO.)    74  All's Well That Ends Well
7              Countess's Page (PAGE.)     1  All's Well That Ends Well
8                    Gentleman (GENT.)    22  All's Well That Ends Well
9     Countess of Roussillion (COUNT.)   220  All's Well That Ends Well
```

I need to remove the notations in parentheses in order to properly join my tables, so I'll use a string method.

[15]:
```
characters[['cter_clean','unneeded']] = characters['cter'].str.split(' \(', 1,␣
 ↪expand = True)
```

[16]:
```
characters.head(5)
```

[16]:
```
                                cter Lines                        Play  \
0               King of France (KING.)   373  All's Well That Ends Well
1             Duke of Florence (DUKE.)    19  All's Well That Ends Well
2  Bertram, Count of Roussillion (BER.)   240  All's Well That Ends Well
3                         Lafew (LAF.)   146  All's Well That Ends Well
4                      Parolles (PAR.)   184  All's Well That Ends Well

                    cter_clean unneeded
0               King of France   KING.)
1             Duke of Florence   DUKE.)
2  Bertram, Count of Roussillion    BER.)
3                         Lafew    LAF.)
```

```
4                    Parolles    PAR.)
```

[17]:
```
#Drop unneeded columns
characters = characters[['cter_clean', 'Play', 'Lines']]
```

[18]:
```
text_play = pd.merge(text, characters, left_on = ['Speaker'], right_on =␣
↪['cter_clean'], how = 'left')
```

This probably achieved a decent amount of matching, but checking the head and tail of the data set reveals two problems:

1) Some lines are spoken by characters with ambiguous names ("All" for the ensemble in response to the First Citizen in Richard III), and so have no match in the character data set.

2) Shakespeare reused some character names across multiple plays. For example, there are characters named Antonio in The Merchant of Venice, Much Ado About Nothing, The Tempest, Twelfth Night, and The Two Gentlemen of Verona, and simply joining on character name is not enough to specify which one of those plays corresponds to that particular Antonio.

Issue (1) has a fairly straightforward-ish solution which should appropriately label many of the NAs. If I assume that these lines are not in a random order, and consist of the full (or partial) dialogue of each play in the order that dialogue is spoken (i.e. a line from Romeo and Juliet is surrounded by other lines from Romeo and Juliet, and is not intermingled with lines from The Taming of the Shrew or Hamlet or King Lear), then I can compare the values of each NA line to the closest non-NA values preceding and following it, and use those to assign a play. For example, line 2, spoken by the ensemble (labeled "All") is preceded by a line identified as matching a character in Richard III, and is followed by a line matching a character in Richard III, and so is almost certainly itself a line from Richard III. I can also fill in the character name with the value from the 'Speaker' column in cases where that would be necessary.

(Note: The example above assumes that the initial match of Richard III for those lines was accurate. I don't think it was accurate, but I'm proceeding as if it was. I'll address this further toward the end of this part.)

[19]:
```
#One column has, for each row, the next non-NA value, while the other has the␣
↪previous non-NA value.
text_play['prev_play'] = text_play['Play'].bfill()
text_play['next_play'] = text_play['Play'].ffill()
```

[20]:
```
import numpy as np
text_play['Play'] = np.where(text_play['prev_play'] == text_play['next_play'],
                             text_play['prev_play'],
                             text_play['Play'])
```

To address issue 2 I described above (the "multiple plays -> 1 character name" problem), I tried the following: since the result of the left join ends up duplicating any rows in the original text data frame which match more than one play, that means that some of the line numbers I generated appear more than once. I can find, for each row, the number of times that line number appears in the data set, set all the values where count(line number) != 1 as NA, and then do the same

comparisons as above.

For example, the selection of rows below shows a conversation between three characters named Gonzalo, Sebastian, and Antonio:

[21]:
```
text_play[9972:9983]
```

[21]:
```
      Line_Number    Speaker  \
9972         7205  Sebastian
9973         7205  Sebastian
9974         7206    Gonzalo
9975         7207  Sebastian
9976         7207  Sebastian
9977         7208    Antonio
9978         7208    Antonio
9979         7208    Antonio
9980         7208    Antonio
9981         7208    Antonio
9982         7209    Gonzalo

                                                      Text  \
9972
An it had not fallen flat-long.
9973
An it had not fallen flat-long.
9974  You are gentlemen of brave metal; you would lift\nthe moon out of her
sphere, if she would continue\nin it five weeks without changing.
9975
We would so, and then go a bat-fowling.
9976
We would so, and then go a bat-fowling.
9977
Nay, good my lord, be not angry.
9978
Nay, good my lord, be not angry.
9979
Nay, good my lord, be not angry.
9980
Nay, good my lord, be not angry.
9981
Nay, good my lord, be not angry.
9982                        No, I warrant you; I will not adventure\nmy
discretion so weakly. Will you laugh\nme asleep, for I am very heavy?

     cter_clean                        Play Lines  \
9972  Sebastian                The Tempest   114
9973  Sebastian               Twelfth Night    98
9974    Gonzalo                The Tempest   135
```

```
9975  Sebastian                  The Tempest   114
9976  Sebastian                Twelfth Night    98
9977    Antonio       The Merchant of Venice   189
9978    Antonio       Much Ado About Nothing    44
9979    Antonio                  The Tempest   142
9980    Antonio                Twelfth Night   105
9981    Antonio  The Two Gentlemen of Verona    35
9982    Gonzalo                  The Tempest   135

                       prev_play                  next_play
9972                 The Tempest                The Tempest
9973               Twelfth Night              Twelfth Night
9974                 The Tempest                The Tempest
9975                 The Tempest                The Tempest
9976               Twelfth Night              Twelfth Night
9977       The Merchant of Venice     The Merchant of Venice
9978       Much Ado About Nothing     Much Ado About Nothing
9979                 The Tempest                The Tempest
9980               Twelfth Night              Twelfth Night
9981  The Two Gentlemen of Verona  The Two Gentlemen of Verona
9982                 The Tempest                The Tempest
```

Sebastian and Antonio each have at least two matches, but Gonzalo only matches one play: The Tempest. As it turns out, Sebastian and Antonio also have The Tempest as a possible match for their dialogue, and since they are talking to a character who only appears in that play, we can assume that these lines correspond to The Tempest, and not one of the other plays which matched the character names.

To do this, I did... well, you'll see. As a result of the left join matching multiple plays to the same line of dialogue, many of the line numbers (like 7208 above) appear more than once in the data set.

First, I used the `value_counts` function to label each row with the number of times that row's line number appears. I then merged the line number counts into the text_play dataframe I've been working with. Then I replaced all the count(line number) values not equal to 1 with `np.nan`, and replaced the values equal to 1 with the corresponding play (to get the play names for the rows with unique matches). I then dropped the `prev_play` and `next_play` columns and re-instantiated them using the column I just constructed with the unique line counts.

Basically, for each line of dialogue which matches multiple plays, this finds the closest unique play before and after it. The same way that I assumed a line where the play was NA but surrounded by lines matching characters in Richard III was itself from Richard III, here I assume that if the closest unique characters to a multiple-matched are from The Tempest, then the correct match from those multiple matches must also be The Tempest, even though that speaker's name appears more than once in the data set.

[22]:
```
line_counts = text_play['Line_Number'].value_counts()
```

[23]:
```
line_counts = pd.DataFrame(line_counts)
```

[24]:
```
line_counts = line_counts.reset_index().rename(columns = {'Line_Number':␣
↪"line_counts",
                                                          'index':␣
↪'Line_Number'})
```

[25]:
```
text_play = pd.merge(text_play,
                     line_counts,
                     left_on = ['Line_Number'],
                     right_on = ['Line_Number'], how = 'left')
```

[26]:
```
text_play['linecounts_play'] = np.where(text_play['line_counts'] != 1,
                                        np.nan,
                                        text_play['Play'])
```

[27]:
```
text_play = text_play.drop(columns = ['prev_play', 'next_play'])
```

[28]:
```
text_play['prev_play'] = text_play['linecounts_play'].bfill()
text_play['next_play'] = text_play['linecounts_play'].ffill()
```

Now I create a new column (creatively titled `Play1`). For a given row, if the previous unique non-NA value of the `Play` column equals the next unique non-NA value, then this column equals that value. Otherwise, it just equals the `Play` column. I also fill in the missing values in the `Play` column with the values in the `prev_play` column, for merging purposes later.

[29]:
```
text_play['Play1'] = np.where(text_play['prev_play'] == text_play['next_play'],
                                        text_play['prev_play'],
                                        text_play['Play'])
```

[30]:
```
text_play['Play'] = np.where(text_play['Play'].isna(),
                             text_play['prev_play'],
                             text_play['Play'])
```

Now, for each value of `Line_Number`, I subset those rows, then create a series containing those rows' values of Play, prev_play, linecounts_play, next_play, and Play1 (basically all of the additional columns and indices I created to try and fix the metadata). Then I find how often each value in that series occurs using value_count, and then drop all the items in the series which occur at the same frequency as the modal value (since I assume that the modal play is correct, a VERY big assumption). Then I append the rows containing the combinations of line number and play title to be dropped to a list (creatively titled `drop_list1`).

Once the for loop below is finished, I then concatenate the list of line number/play combos I want to drop, convert it to a data frame, and then performan an anti-join between the text_play dataframe and that dataframe.

[32]:
```
drop_list1 = []
for ii in text_play['Line_Number'].unique():
    #if text_play1[text_play1['Line_Number']==ii]['line_counts'].unique() != 1:

    #Select rows where line number = ii
```

```
    rows = text_play[text_play['Line_Number']==ii]

    #Select these 5 columns, and stack them into a single series
    options = rows[['Play',
                    'linecounts_play',
                    'prev_play',
                    'next_play',
                    'Play1']].stack()

    #Find the value counts and set them up as a dataframe
    selection = pd.DataFrame(options.value_counts().reset_index()).
↪rename(columns = {'index': 'Play',
                                                                    ␣
↪         0: 'count'})
    #If there is more than 1 pair of line number and play, drop the rows with␣
↪the max frequency
    if len(selection) > 1:
        to_drop = selection[selection['count']!=selection['count'].max()]
        #Add a column of just the line number for the anti join
        to_drop['Line_Number'] = ii
        #Append to the drop list
        drop_list1.append(to_drop)
        #to_drop = to_drop[['Line_Number', 'Play']].apply(tuple, axis=1)
        #text_play1 = text_play1[~text_play1[['Line_Number','Play']].
↪isin(to_drop)]
```

```
<ipython-input-32-0a192a9eb097>:21: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  to_drop['Line_Number'] = ii
```

[33]:
```
drop_list1 = pd.concat(drop_list1)
```

[36]:
```
text_play1 = text_play.merge(drop_list1, on = ['Line_Number', 'Play'], how =␣
↪'left', indicator = True)
```

[37]:
```
text_play1 = text_play1.loc[text_play1['_merge'] == 'left_only']
```

This is the part I don't fully understand. The dataframe I produce at the end of part 1 has 7,222 rows, and theoretically, the data frame I produce at the end of part 2 should have that many rows as well. However, the dataframe `text_play1`, which is my final product for part 2, only has 6987 rows. Granted, if each row (meaning each chunk of text) is a "document", then that represents a loss of approximately 3.2%. Not great, but I don't think that's a project-ending level of loss.

Side note: if you can figure out where in this process I lose those 235 rows, please let me know!

The last note I have on this particular part is that while this process achieves a reasonable degree of matching, I know that not all of the matches are accurate. This is due to a mismatch between the speaker identifiers in the original data and the identifiers used by playshakespeare.com. This data set might list a character as "First Citizen" and have it correspond to "First Roman Citizen" in the play Coriolanus, while the character playshakespeare.com identifies as "First Citizen" is actually from the play Richard III. Looking back on what I did for this part (which was largely improvised), I think I tried to apply some of the same assumptions and ideas we discussed in class (mainly a super-generalized Markov assumption that things near each other are likely to be more similar than things farther apart), but didn't end up doing a lot of *text*-based matching.

Given more time, I probably could have achieved a more accurate set of matches between text and play using the data available. I tried to use a list comprehension method to match character names from the character dataset to lines of dialogue, but was unable to get that to work (something to do with a list not being hashable).

---

8 Part 3

Pick one or more of the techniques described in this chapter:

```
keyword frequency
entity relationships
markov language model
bag-of-words, TF-IDF
semantic embedding
```

and make a case for a technique to measure how important or interesting a speaker is. The measure does not have to be both important and interesting, and you are welcome to come up with another term that represents "useful content", or tells a story (happiest speaker, worst speaker, etc.)

Whatever you choose, you must

```
document how your technique was applied
describe why you believe the technique is a valid approximation or exploration of how important,
list some possible weaknesses of your method, or ways you expect your assumptions could be viola
```

This is mostly about learning to transparently document your decisions, and iterate on a method for operationalizing useful analyses on text. Your explanations should be understandable; homeworks will be peer-reviewed by your fellow students.

First, I select only a few columns from the data set (I don't need a lot of the columns of plays I used and generated in part 2 for this step).

[51]:
```
text_play2 = text_play1[['Line_Number','Speaker','Text','Play']]
```

It looks like there are 125 rows which are missing text. I'm not sure why these rows are missing, but there's no way to perform text analysis on missing text, so I'll just drop those.

[54]:
```
text_play2 = text_play2.dropna()
```

Next, I concatenate the text so all the dialogue from the same character in the same play becomes a single string instead of several strings.

[55]:
```
text_play2['text_concat'] = text_play2.groupby(['Speaker','Play'])['Text'].
↪transform(lambda x: ','.join(x))
```

Next, I drop duplicates, so each Speaker-Play pair only appears once in the dataset.

[57]:
```
text_play3 = text_play2.drop_duplicates(subset=['Speaker','Play'])
```

We've discussed its limitations in class, particularly with respect to the nature of the tokenizer, but I choose to use scikit-learn to implement TF-IDF (term frequency-inverse document frequency) for this part. The scikit-learn tokenizer might not be effective for all document types or text sources, but for this case, where the corpus consists of standard English literature, the scikit-learn tokenizer is perfectly adequate.

TF-IDF is a measure of term significance. It is proportional to the frequency of a token in a document, and inversely proportional to the frequency of the term across all documents. A token like "the", which is highly prevalent in many texts, would have a low TF-IDF score, even though it is common. Conversely, a token like "TFIDF", which only occurs in a small corpus of documents (namely, those regarding text analysis and NLP) but is highly prevalent in those specific documents might have a high TFIDF score.

[127]:
```
from sklearn.feature_extraction.text import TfidfVectorizer, TfidfTransformer,␣
↪CountVectorizer
```

Now I instantiate a vectorizer, fit it to the series of dialogue strings, and then convert it to a data frame.

[157]:
```
vect = TfidfVectorizer()
tfidf_matrix = vect.fit_transform(text_play3['text_concat'])
df = pd.DataFrame(tfidf_matrix.toarray(), columns = vect.get_feature_names())
```

I calculate the mean TF-IDF score for each Speaker-Play pair:

[158]:
```
df['mean_TFIDF'] = df.mean(axis = 1)
```

Then I add in the speaker and play as identifiers, and select them and the means.

[159]:
```
df_TFIDF = pd.concat([text_play3[['Speaker','Play']].reset_index(drop=True),␣
↪df], axis=1)
```

[161]:
```
df_TFIDF = df_TFIDF[['Speaker','Play','mean_TFIDF']]
```

Calculating the mean TF-IDF score for each character-play pair will show who uses the most unusual or distinctive words in this corpus of Shakespeare plays.

[181]:
```
df_TFIDF.sort_values(['mean_TFIDF'], ascending = False).head()
```

[181]:

	Speaker	Play	mean_TFIDF
301	Leontes	The Winter's Tale	0.001835
157	King Richard Ii	King Richard II	0.001829
217	Romeo	Romeo and Juliet	0.001822
221	Mercutio	Romeo and Juliet	0.001781
26	Coriolanus	Coriolanus	0.001781

The character Leontes, from the play The Winter's Tale, has the highest TF-IDF score. Granted, this, and the other analyses done in this part, are predicated on the idea that the dialogue was appropriately matched to a play in part 2 (though we know that this line was spoken by Leontes, since the speaker identifiers were included in the original data).

[163]:
```
df_TFIDF[df_TFIDF['mean_TFIDF']==df_TFIDF['mean_TFIDF'].min()]
```

[163]:

	Speaker	Play	mean_TFIDF
58	All	Double Falsehood	0.000089
64	Both Tribunes	Richard III	0.000089
167	All	King Richard II	0.000089
331	Duke Vincentio	The Winter's Tale	0.000089
365	All	The Taming of the Shrew	0.000089
392	Master	The Taming of the Shrew	0.000089

There is a six-way tie for lowest mean TF-IDF score. Notably, four of these speakers are ensembles: the two "All"s and "Both Tribunes". This likely reflects the fact that in many Shakespeare plays, the ensemble dialogue mainly consists of short exclamations or invocations for main characters to deliver monologues.

[168]:
```
tfidf_plays = df_TFIDF.groupby(['Play']).mean(['mean_TFIDF'])
```

[170]:
```
tfidf_plays.sort_values('mean_TFIDF')
```

[170]:

Play	mean_TFIDF
Henry VIII	0.000280
Titus Andronicus	0.000316
Double Falsehood	0.000503
Timon of Athens	0.000558
Twelfth Night	0.000626
Henry VI, Part 2	0.000679
Coriolanus	0.000703
Sir Thomas More	0.000727
The Taming of the Shrew	0.000740
Richard III	0.000776
Henry VI, Part 1	0.000777
Measure for Measure	0.000851
Romeo and Juliet	0.000870
King Richard II	0.000898
Henry VI, Part 3	0.000912

```
The Tempest                0.000928
The Winter's Tale          0.000929
Edward III                 0.001009
Cymbeline                  0.001088
```

Based on these results (and again, assuming that dialogue was mostly matched to the correct play), Cymbeline, Edward III, and The Winter's Tale are among the most distinctive of Shakespeare's plays, while Henry VIII, Titus Andronicus, and Double Falsehood are among the least distinctive. I tried to identify a pattern based on the chronology of the plays https://en.wikipedia.org/wiki/Chronology_of_Shakespeare%27s_plays, but none is evident; Cymbeline was one of Shakespeare's last plays, but so was Henry VIII.

[172]:
```
values = df_TFIDF['Play'].value_counts()
values = pd.DataFrame(values)
values
```

[172]:
```
                         Play
Richard III                54
Romeo and Juliet           44
Coriolanus                 36
The Taming of the Shrew    35
The Winter's Tale          31
Henry VI, Part 3           30
King Richard II            29
Sir Thomas More            27
Measure for Measure        26
Henry VI, Part 1           17
Henry VI, Part 2           15
The Tempest                14
Cymbeline                  12
Timon of Athens            12
Twelfth Night              11
Double Falsehood            9
Titus Andronicus            3
Henry VIII                  1
Edward III                  1
```

There also does not appear to be a significant relationship between the number of characters identified as being part of a particular play and the mean TF-IDF score of that play. However, the play Edward III had one of the highest mean TF-IDF scores, and only one character from that play is identified in this data set.

[173]:
```
text_play3[text_play3['Play']=='Edward III']
```

[173]:
```
      Line_Number        Speaker  \
2258         1568  Prince Edward

                                                  Text  \
```

```
2258  No, uncle; but our crosses on the way\nHave made it tedious, wearisome,
and heavy\nI want more uncles here to welcome me.

            Play  \
2258  Edward III

                                        text_concat
2258  No, uncle; but our crosses on the way\nHave made it tedious, wearisome,
and heavy\nI want more uncles here to welcome me.,God keep me from false
friends! but they were none.,Methinks a woman of this valiant spirit\nShould, if
a coward heard her speak these words,\nInfuse his breast with magnanimity\nAnd
make him, naked, foil a man at arms.\nI speak not this as doubting any here\nFor
did I but suspect a fearful man\nHe should have leave to go away betimes,\nLest
in our need he might infect another\nAnd make him of like spirit to himself.\nIf
any such be here--as God forbid!--\nLet him depart before we need his help.,And
take his thanks that yet hath nothing else.,Speak like a subject, proud
ambitious York!\nSuppose that I am now my father's mouth;\nResign thy chair, and
where I stand kneel thou,\nWhilst I propose the selfsame words to thee,\nWhich
traitor, thou wouldst have me answer to.,Let AEsop fable in a winter's
night;\nHis currish riddles sort not with this place.,Nay, take away this
scolding crookback rather.,I know my duty; you are all undutiful:\nLascivious
Edward, and thou perjured George,\nAnd thou mis-shapen Dick, I tell ye all\nI am
your better, traitors as ye are:\nAnd thou usurp'st my father's right and mine.
```

That character is Prince Edward (listed on Wikipedia as Edward the Black Prince), the eldest son of King Edward III and the heir apparent to the throne until his death in 1376. Just looking at this dialogue, several unusual words stand out: magnanimity, valiant, selfsame, currish, crookback, lascivious, perjured and usurp'st all make appearances in this text. This is likely an excerpt from a dramatic monologue, maybe the one where he almost loses in battle to the French. This might be going a bit too deep down the Shakespeare rabbit hole, but the Wikipedia page for Edward III presents theories and research by Shakespeare experts which cast doubt on the idea that he even wrote this play, or at least that he was the sole author.

Having acknowledged this, it is necessary to point out that Prince Edward does not rank in the top 100 characters in this data set for highest mean TF-IDF score. Across all the plays included, there are 407 characters; as such, while Prince Edward is in the top 25%, he is by no means an exception with respect to unusual vocabulary. Examining the character with the most distinctive dialogue, Leontes, is revealing:

[Text removed for brevity. The full length of dialogue attributed to Leontes in this data set spans four pages, and can be viewed in the notebook submitted with this assignment,]

The length of dialogue attributed to this character is significantly longer than that attributed to Prince Edward. This reveals a potential weakness of TF-IDF: difficulty handling different document sizes. Leontes might not have a more interesting or idiosyncratic vocabulary than Prince Edward; he might just talk more. Future steps for this analysis could weight TF-IDF scores by document size, or could select a random sample of tokens from each document to use to calculate scores.

That being said, I do think that TF-IDF is useful as a means of evaluating terms and documents in

relation to each other. This metric makes it easy to identify the most distinctive terms and tokens in a document, and also helps identify documents which could be outliers in the corpus. In this case, I found that one of the plays with the highest mean TF-IDF scores was potentially not written by Shakespeare! Additionally, since the sole character we have data on in that play had a TF-IDF score which was only in the top 30% of characters in the data set, this suggests that many other characters, like ensembles or parts which only appear in one or two scenes, have relatively generic dialogue or do not use unusual or uncommon words. TF-IDF can also be implemented relatively easily, without having to download a large model or data set, making it easier and quicker to use than other processing methods such as embeddings.