Kevin Xie

Final Project 2 Report

PIC16

I. The Problem

A. Intro

The goal of the project is to display a network of characters from a story in an easy-to-read network plot. There are four salient problems one must address to fully execute this task. One is to figure out who exactly the characters are in the story. This involves some principles which form the backbone of a field in computer science known as natural language processing. This field is incredibly complex and involves copious amounts of machine learning. If the reader is interested in this field, he/she may look at the Natural Language Toolkit for reference. As the details are far beyond this course, we used some tricks to try to work around this issue. Two is to figure out how to determine how related two characters are in a story and how to best convert this relationship into a standard data structure. Three is to rank the characters. Four is to figure out how to create a force-directed graph drawing algorithm to present the data in a natural and aesthetically pleasing way. The method I settled on involves polar coordinates and proximity to display magnitude of connection. Ultimately, my project was tested on two books: *Harry Potter and the Sorcerer's Stone (2001)* and *The Lord of the Rings: The Fellowship of the Ring (2001).*

B. Character Discovery

Let us first address how to discover characters in a string. This is perhaps the most arduous part of the whole project. Many online PDFs have an inordinate amount of typos and so when one comes up with a rudimentary system to catch proper nouns many other strange words come with it. Nonetheless the system I used was to first transfer all the text in the PDF into a .txt file, and then split up the text by periods. Within these sentences the first word was always ignored. I then used a Regex expression re.findall(r'(?:[A-Z][a-zA-Z\-\.]+)*[A-Z][a-zA-Z\-\.]+',sentence) to search for a string of words in which each word begins with a capital letter; each string of words is then appended to a list. After parsing through the entire story the list goes through three filtering processes. The first process tries to eliminate words commonly capitalized in the beginning of

dialogue/monologue: ['Good-bye','Take','Chapter','Now','Before','Could','Mr','Mrs','Men','What','Too','Here','That','Then','When','This','The','It','We','You','With','Her','Him','She','He','They','Them','Why','Who','How','An','Yes','No','And','But','So','Me','There','For']. The second process tries to eliminate character dialogue which involves shouting. Often this manifests in a completely capitalized word like 'NO'. So if i.upper() != i with i being the word of interest the word is allowed to remain in the list. There are an unfortunate amount of cases where characters will often shout another character's name but I argue this second process easily does proportionally more good than bad. The third and final process looks at the length of the discovered word and removes all words whose length is less or equal to two. I settled on two because names minimally have three letters like "Sam", "Ash", and "Max" and rarely go below this threshold. The full pseudo-code for my described function can be found in the last section written under **def findProperNouns(self,sentences)**. The words belonging the list outputted by the function will have many redundant names. A list(set(self.findProperNouns(sentences))) removes these. The list is now **self.names.**

C. Transfer into Data Structure

Let us now discuss how to represent these characters in a data structure. The full pseudo-code for the following instructions is detailed in **def createMatrixForPlotting(self,sentences)**. We start by creating a 2D numpy array of height and width the length of the **self.names**; so essentially we have enumerated each name in this step. We take our **self.names** and go through each sentence again and count how many times the names appear in the sentence. One should create a list to keep track of the names in the sentence and then reform the list with the index values from **self.names**. There are a few ways to do the next part. The method I chose is to find the minimum index, remove it from the index list, and then go the integer's respective column and row, adding one to any entry associated with the remaining indices. This should give you a basic weighted symmetric adjacency matrix. Now, like in the previous process, we must filter out obscure and superfluous entries. I chose two processes to eliminate columns and rows. The first process is rather simple: eliminate any row or column which has nothing but zeros. For the second process, I created a concept known as the specificity constant. For any index, if the row or column never has any entries above this specificity constant we eliminate the row or column. The user should note that 2 is an ideal specificity constant value for both Harry Potter and Lord of the Rings. In actuality, the second process covers the first process but since I am representing the weighted symmetric adjacency

matrix as a 2D numpy array, I can utilize numpy's sum feature and gain a higher performance by skipping the first process' looping. I did not include the pseudo-code for this as the function is rather straight forward. The reader may evaluate the code on his/her own: **def cleanseMatrix(self,N)**. The matrix we get from this process is called **self.network**.

D. Ranking

Before we analyze the graph drawing portion the reader should understand how the ranking portion was done. It is a standard PageRank algorithm which utilizes the numpy linear algebra module. It is detailed in this report http://www.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture3/lecture3.html. The method involves two steps. First is making the column entries add up to one: this involves summing up the network row, dividng each entry by the sum, and then transposing it. With numpy, this task is incredibly convenient and easy. The function used to complete this process is called **def makeColumnsSumTo1(self)** and the variable it creates is called **self.matrixColumn1**. Second is finding the eigenvector associated with the eigenvalue of one. This eigenvector contains values which can be ranked to find out which character is referenced the most often and most significantly. The eigenvectors are then mapped to the associated index and sorted, leaving only the indexes sorted by PageRank. The function which does this process is called **def pagerank(self,N)** and the list it produces is called **self.ranks.** The reader may evaluate the functions discussed in the attached python file.

E. Graphing

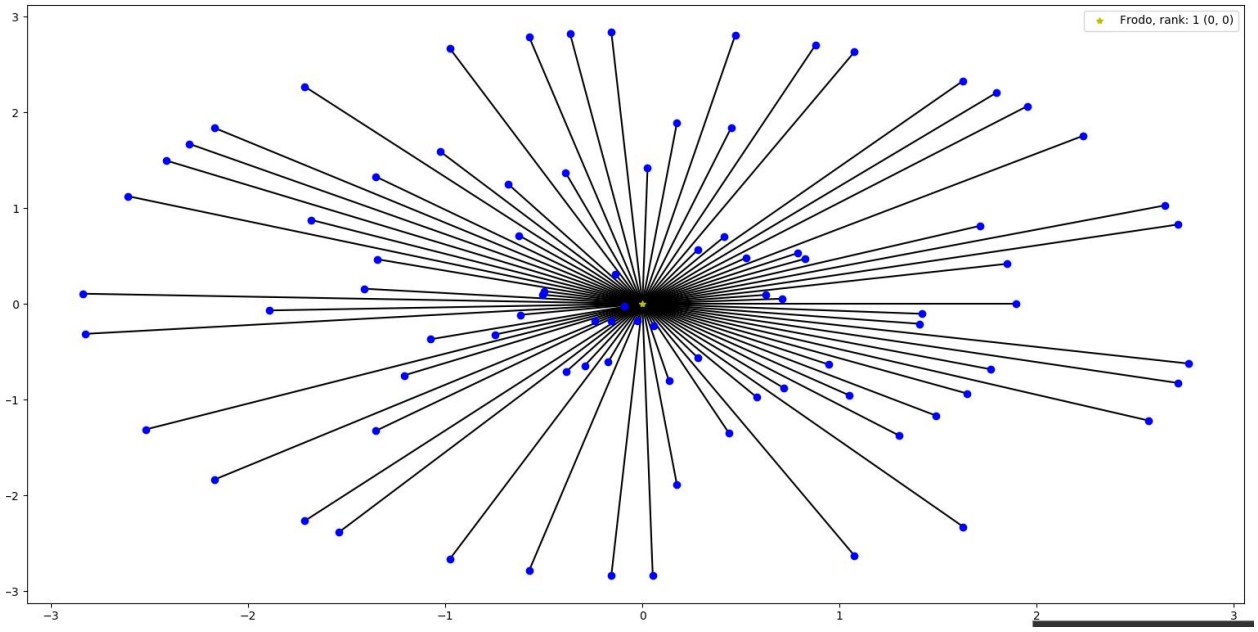
Finally, let us analyze the graph drawing algorithm used to connect edges between nodes. The algorithm is based off a method used in mathematics called the midpoint formula where one can find a coordinate between two points i.e. if x = (x1,y1) and y = (x2,y2) then the midpoint is ((x1+x2)/2,(y1+y2)/2). Of course, we don't want to just find the midpoint between points. From https://math.stackexchange.com/questions/563566/how-do-i-find-the-middle1-2-1-3-1-4-etc-of-a-line, I discovered that I can go a portion of the way between two points. For example, a third of the way from point x my algorithm would change to: ((2/3)x1+(1/3)x2,(2/3)y1+(1/3)y2. This formula can be generalized to become: ((1-factor)x1+(factor)x2,(1-factor)y1+(factor)y2). This factor is incredibly important to the algorithm and is exactly the entry value of a transposed

self.matrixColumn1. A note to the reader, I use a transposed **self.matrixColumn1** because it's more intuitive for me to retrieve an entry but this step is not necessary.

Let us discuss the method used to scatter nodes. First off, the node with the highest rank is selected to form the center of the graph which for my project is exclusively in the origin. This is the first value in our pagerank list. We look at the row for this value and pick out each entry that has a value higher than the specificity constant we discussed earlier and save the indexes; these indexes become our nodes. A factor list is created by reading off the index entries in the transposed **self.matrixColumn1**. The nodes are then scattered in a circular pattern based on polar coordinates and the said factor list.

x=[factor[k]*np.cos(2*np.pi*k/n) for k in range(n)]

y=[factor[k]*np.sin(2*np.pi*k/n) for k in range(n)]

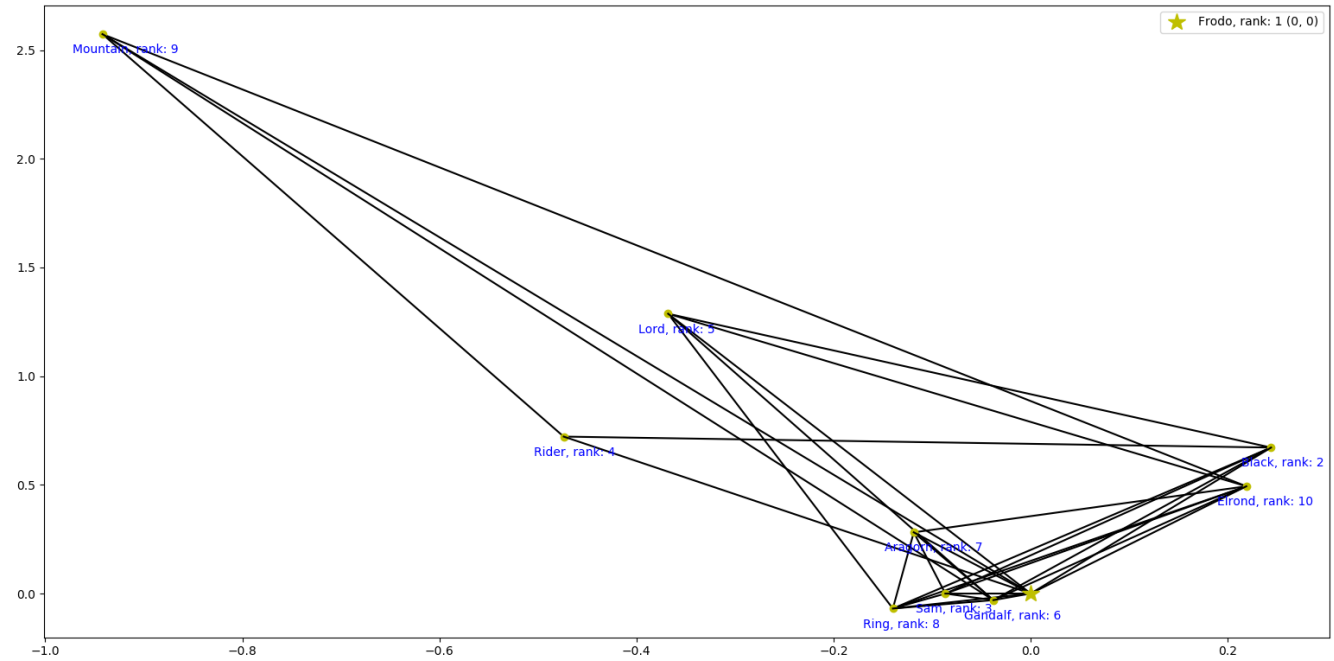


We then iterate through ever node and compare that node to the other nodes. In two for loops, if an entry exists in the transposed self.matrixColumn1 for those two nodes we create an edge and then change the node coordinates based on the factor entry value. We do both ((1-factor)x1+(factor)x2,(1-factor)y1+(factor)y2) and ((1-factor)x2+(factor)x1,(1-factor)y2+(factor)y1)

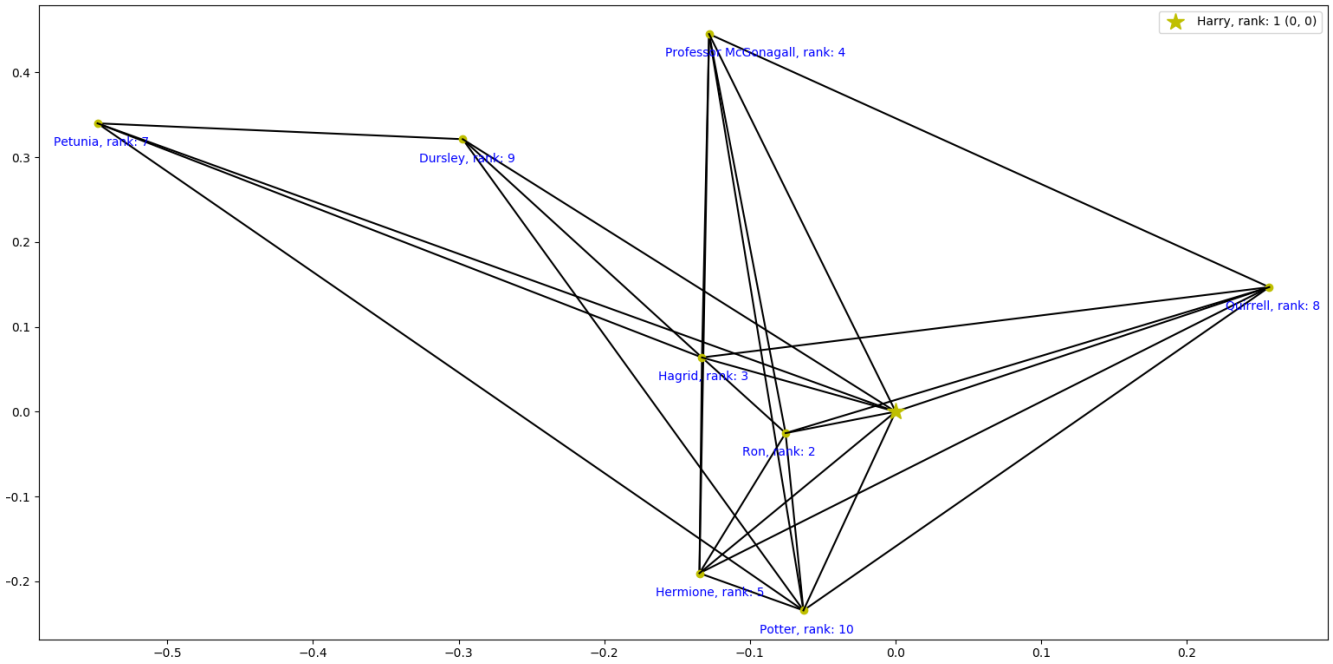
to change both positions of the nodes of interest. The closer the factor is to 0.5, the more close the two nodes are drawn to each other The full pseudo code for the function which describes this process is called **def network_plot_circle(self,title)** and is located in the last section.

II. Results and Improvements

The Lord of the Rings: The Fellowship of the Ring (2001) ------- character_network(lord,10,2)



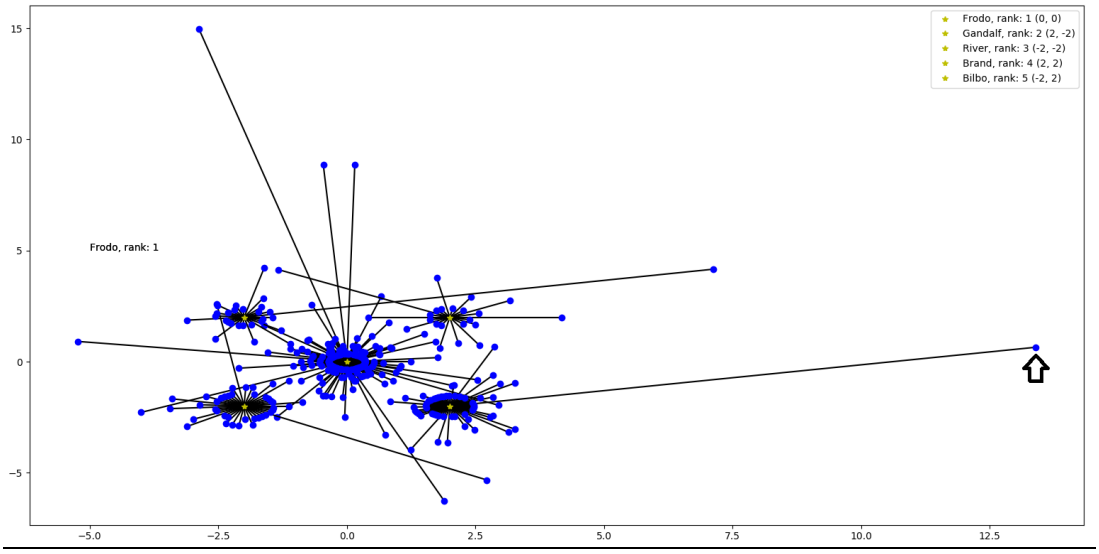
Harry Potter and the Sorcerer's Stone (2001) ------- character_network(harry,10,2)



Note, the second parameter 10 for both object calls represents the max number of characters that can be connected to the central node. As the reader can see, the originally circular format is now completely changed with characters more connected being closer together. The graph overall shrinks a little bit. My specificity constant and basing all my data on how far the most popular character reaches removed some nodes out of the picture. Furthermore, one of the things I could not perfectly execute was the annotations. It is particularly messy where all the very connected characters are. I tried to implement a click option for the nodes as described by this Stackoverflow post https://stackoverflow.com/questions/7908636/possible-to-make-labels-appear-when-hovering-over-a-point-in-matplotlib but for some odd reason my program crashed continually and I could not get it to work perfectly i.e. I could make the name and rank appear but the name and rank would disappear and the plot would crash.

Apart from this there are three main things the reader can improve on. One is giving the user a choice on who to center the network around. This can be as simple as asking for a input name in the beginning of the program. Two is utilizing a Natural Language Processing module for the character discovery portion. There are a few strange nouns that appear like Mountain in the top left corner of the LOTR graph which sully the overall legitimacy of the presentation. The third and final thing is work on the presentation in a way that will fit more characters. The max character parameter was necessary for my limited presentation skills but looking online there are ways to do a 3D graph which may be more appropriate for the project.

III. Failed Attempts



I created a cluster based graph which spaces the most popular characters out. This design is based off the preliminary scattering discussed in the graphing portion. It is however the opposite of a typical force-directed graphing algorithm in that the more popular a character is the farther the node will stretch. This graph however does not connect the clusters in any way and that was its primary flaw. If the reader is interested in the code he/she may visit the attached python file.

IV. Pseudo-Code

All of the following functions are under a class called character_network. The member variables are:

self.story = story

self.names = []

self.network = [[]]

self.matrixColumn1 = [[]]

self.ranks = []

self.maxNOC = maxNumberOfCharacters

self.sC = specificityConstant

These member functions are discussed in previous sections

def findProperNouns(self,sentences): # finds proper nouns

create an empty list called properNouns

for every sentence in sentences

split the sentence by spaces

if the first value in the list has a empty white space

cut off the first and second values in the list

else:

cut off the first value in the list

rejoin together the list through spaces into a string

properNouns.extend(re.findall(r'(?:[A-Z][a-zA-Z\-\.]+)*[A-Z][a-zA-Z\-\.]+',sentence))

properNouns = [i for i in properNouns if i not in self.wordsToRemove and i.upper() != i and len(i) > 2]

return properNouns

def createMatrixForPlotting(self,sentences):

create a NxN numpy array matrix where N is measured as the length of self.names called superMatrix

for every sentence in sentences:

create an empty list called listOfNamesForSentences

for every name in the self.names:

if name in sentence

listOfNamesForSentences.append(name)

indexList = find all the indexes for the names in listOfNamesForSentences

if there are list elements in indexList:

find minimumIndex and remove it from indexList

for value in indexList:

add one to superMatrix [minimumIndex][value]

add one to superMatrix [value][minimumIndex]

return superMatrix

def network_plot_circle(self,title, numberOfCharacters, specificityConstant):

initialize a plot

rowColumn1 = transpose self.matrixColumn1

characterPoints = empty dictionary

popularCharacterIndex = self.ranks[0] (most popular character)

nonZeroIndexList = go to self.network[popularCharacterIndex] and get all values in that list above the specificity constant

factor = go to rowColumn1[every value in nonZeroIndexList]

```
factor = [mean of factor /z for z in factor]
x=[factor[k]*np.cos(2*np.pi*k/n) for k in range(n)]
y=[factor[k]*np.sin(2*np.pi*k/n) for k in range(n)]
for every integer from 0…len(factor)
    if the rank of (nonZeroIndexList[z]) < max number of charachters:
        characterPoints[nonZeroIndexList[z]] = (x[z],y[z])
pointConnections = empty dictionary
dicList = list of characterPoints keys
for i in 0….len(dicList)-2
    networkI = dicList[i]
    for j in i+1….len(dictList)-1
        networkJ = dicList[j]
        factor = matrixRow1[networkI,networkJ]
        if factor != 0:
                append networkJ to the list pointConnections[networkI]
            x1 = characterPoints[networkI][0]
            y1 = characterPoints[networkI][1]
            x2 = characterPoints[networkJ][0]
            y2 = characterPoints[networkJ][1]
            newx = (1-factor)*x1+factor*x2
            newy = (1-factor)*y1+factor*y2
            characterPoints[networkI] = (newx,newy)
            newx = (1-factor)*x2+factor*x1
            newy = (1-factor)*y2+factor*y1
            characterPoints[networkJ] = (newx,newy)
```

----- At this point all of your data is stored, it is up to you on how you want to graph it--------