CSM6720: Applied Data Mining Project Report

Jamie Hall / jah79

Introduction

This report will cover the program and code used in the project for the data mining module assignment. The purpose of this report is to give the reader a better understanding of the program and code used within it. There will be insight into the problems tackled, issues that arose and the solutions used as a result. This report will explain the decisions made throughout the project and should help the reader understand why they were made.

Problem Overview

The customer is part of a fictional organisation known as the Local Heritage Foundation which have an interest in the history of ships that visited Aberystwyth and the crews that manned those ships. They have asked for the ability to trace the lives of each sailor as they progress through their careers and see how many ships they had worked on in their career lifetime. Sailor promotions also need to be traceable. Data on ships, sailors and the ports they visit must be viewable in some form to allow the user to see which ports were most popular, which ships visited and which crew members were onboard.

As this was a broad request, the task was broken down into sub-tasks. The tasks are as follows:

1) Part One – Individual Stories
    a) List all sailors for the user to see what they can select
    b) Retrieve all records on a selected sailor
    c) Visualise the number of sailors at each rank
    d) Visualise the promotion track of two or more sailors

2) Part Two – Who is Visiting
    a) Look at ship records and extract ports
    b) Visualise number of crew for each ship as a histogram
    c) Number of unique ship visits to Aberystwyth port

A database was provided with the data to be used for the task. This database included ship details, sailor details and port information.

Literature review

Upon reading the assignment brief, I looked up an article on similar issues I would dealing with for this project. The article read was ‘Duplicates, redundancies and inconsistencies in the primary nucleotide databases: a descriptive study’ (Chen, Zobel and Verspoor, 2019). This lengthy article provides good insight into how much duplication, redundancy and inconsistency can negatively affect the accuracy of sometimes important results. Although the database looked at within this article is bioinformatics database, the same issues affect databases holding all kinds of information. The article also states it affects the accuracy more when it’s used in an aggregation or summary. Forms of aggregation and summaries are used throughout this project and this shows that this

problem for the bioinformatics database can also be the same problems we have to deal with as a result of the state of this database.

The data

As preparation for the project, I looked at the data within the database to see what I was dealing with. One of the fields (mariners), appears to be a 2d array containing fields and data on the sailors. There are many misspellings, same word variations and missing data throughout the dataset that will need to be dealt with appropriately. Below are the notes made upon analysis of the database fields and data:

Attribute	Data information
_id	Uniquely identifying each record.
vessel name	Contains the name of a ship, not unique as same ships appear multiple times throughout this field.
official number	A ranking number? Appear to be a number which is the official number of the record from when it was entered.
port of registry	The port at which the ship was registered at.
mariners	This field is an array with sub-fields within. It contains data about the sailors. Sub-fields listed below.
mariners: name	The name of the sailor.
mariners: year_of_birth	The year of birth for the corresponding sailor.
mariners: age	The age of the sailor.
mariners: place_of_birth	The location at which the sailor was born.
mariners: last_ship_name	The name of the last ship the sailor was serving on. More misspellings found and missing data.
mariners: last_ship_port	The port at which the sailor left their last ship. Further misspellings and blank records. This appears to be a major issue throughout the database.
mariners: last_ship_leaving_date	The date of when the sailor left their last ship.
mariners: this_ship_joining_date	The date of when the sailor joined the ship they are currently serving.
mariners: last_ship_joining_port	The port at which the sailor joined their current crew.
mariners: last_ship_capacity	The rank of the sailor.
mariners: this_ship_leaving_cause	The reason for leaving the current ship. Although, they may not have left yet, in which case there is then either a note stating the sailor is still onboard or the field is left blank/blk.
mariners: signed_with_mark	What appears to be a signing confirmation of the records with yes or no.
mariners: additional_notes	Additional notes made about the sailor.
mariners: this_ship_leaving_port	The port where the sailor left this ship, if they have left this ship.

Issues with the data

From Looking at the various attributes and the data held within them, there appears to be a major issue with data consistency in this database. There are frequent misspellings throughout the database and a large amount of missing data. Ideally, it would likely be best to manually re-do the database with input validation and verification to ensure the data entered was correct and that missing data could be dealt with during the input phase. However, as this is the state of the database

the customer has requested to work with, the missing data and misspelling of words must be dealt with in the code.

The dates stored seem to follow a variety of formats with some only recording year, some recording both month and year whereas others seem to follow a datetime format. This may prove an issue should the date fields be used.

Methodology

This section looks at the separate functions of the program, what they solve, and any decisions made on the code within them.

Listing all unique/individual sailors

Initially a distinct() function was used to find all distinct/unique name instances but it was later noticed that there are some sailor records of the same sailor which have an added middle name or name variation. To account for this, an aggregate was used instead of the distinct function. This aggregate takes the first name of the field and the last name of the field and forms a sailor's name based of those two words (forename and surname).

```
sailor_docs = db.jah79.aggregate([{"$unwind": "$mariners"},
                                  {"$sort": {"mariners.name": 1}},
                                  {"$group": {"_id": "$mariners.name"}
                                   }])
```

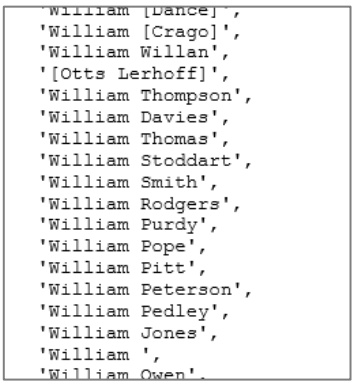
Some records also appear to only have initials for the first name (e.g. E Jones) this could have been solved by only looking at the first letter of the field, but this could cause incorrect results to be returned as there may be an 'Eric Jones' and an 'Edward Jones' with the same year of birth, by coincidence. Admittedly this problem may still happen by using the first name and last name, but it is less likely to be an issue at least. A balance was found between the issue of incorrect records being retrieved and not retrieving ALL the sailor's records by using the first name and last name.

A for loop was used to start storing each letter of a name from the returned document until a space was found, then the for loop would stop. This would give the first name and a similar process was used to find the last name although working from the end of the field back.

```
#Uses returned doc as a string to dissect the required data
count = 0
sailors = []
for doc in sailor_docs:
    count = count + 1
    stringdoc = str(doc)
    #Removing unnecessary/irrelevant characters
    namestring = stringdoc[9:-2]

    #Using for loop to add each character from string until a space is reached
    firstname = ""
    for i in range(len(namestring)):
        if namestring[i] == " ":
            break
        else:
            firstname = firstname + namestring[i]
```

The year_of_birth field is used to differentiate sailors with the same name as it is unlikely two sailors with the same name would also have the same birthday, but it is possible. unfortunately, there are no other uniquely identifying fields to find all unique sailors due to the variation in a sailors' record field. As some year_of_birth values are missing. The age could be used as a possible replacement value for that sailor, but some records of a sailor may produce missing age and provide a year of birth but in other instances of the same sailor an age may be provided, and the year of birth could be missing instead. This makes it near impossible to settle on a good or even useful method to accurately differentiate sailors.



The results are stored in a file and the file name is displayed to notify the user. The decision to save the results as a file is solely due to how difficult it would be for the user to look through all the results through a command prompt, a text document makes it considerably easier to look at.

Extracting all records on a sailor

A function was used to contain the code that solves this task. The function requires input from the user requiring details of the sailor they wish to retrieve the records of, this includes the first name, last name and year_of_birth. The input data is then used in a more complex aggregate function as shown below:

```
cursor = db.jah79.aggregate([{"$match": {"$text": {"$search": str(searchkey)}}},
                             {"$unwind": "$mariners"},
                             {"$sort": {"mariners": 1}},
                             {"$match": {"mariners.year_of_birth": int(s_year)}}
                             ])
```

This aggregate uses an initial text search to find only the records where the mariners.name field contains both the first name and the last name entered by the user. To search through the names field in a text search, an index first had to be created with the mariners.name field. This was done with the following code:

```
cursor = db.jah79.create_index([("mariners.name", pymongo.TEXT)])
```

Some issues were encountered when trying to use two variables in a text search and making sure the search returned records which include both words and not records which only contain one of the names. The issue was overcome by using the following code to let the text search know these are two separate words that both must be within the field:

```
searchkey = "\"" + str(s_first_name) + "\"\"" + str(s_last_name) + "\""
```

The next step in the aggregate involves the 'mariners' records being broken down with unwind and sorted alphabetically. A final match is then used to only include the results which also include the same year_of_birth that the user entered.

The resulting records are then output as a text file for the user to view and use as they please. Again, command line was not used here as it would be awkward for the user to look through the resulting records.

```
["{'_id': ObjectId('5cbd1f96b58a4409462f7ba6'), 'vessel name': 'Rheidol',
"
 "'official number': 67643, 'port of registry': 'Aberystwyth',
'mariners': "
 "{'name': 'Edward Jones', 'year_of_birth': 1855, 'age': 'No info', "
 "'place_of_birth': 'Liverpool', 'home_address': 'No info',
'last_ship_name': "
 "'Naval Reserve', 'last_ship_port': 'Liverpool',
'last_ship_leaving_date': "
 "1877, 'this_ship_joining_port': 'Liverpool', 'this_ship_capacity': '$',
"
 "'this_ship_leaving_date': '01/03/1879', 'this_ship_leaving_port': "
```

Although not aesthetically pleasing to look at and not easy to search through, the correct data is displayed, and the file allows the user to print-off the results or use in other programs.

Extracting all unique records from the database

This function was initially created for the visualisation sections of the assignment, but the decision was made to make this function available to the user too in case they wish to retrieve all unique records saved in the database and save them to a text file. This could be used for the user to print and hold a paper copy for possible back-ups or other reasons and would provide a version of the database without duplicates if there are any.

To do this, a simple distinct() function was used on mariners so all unique mariners records are returned, excluding duplicates if there are any.

```
if search_type == "2":
    cursor = db.jah79.distinct("mariners")
    filename = "all_records"
```

Visualisations of rank numbers

Matplotlib.pylot and numpy libraries were added as their features were required. I initially tried utilising $toLower within the aggregate but failed to get it to work after countless attempts and little help found online. This would have solved the issue where ‘Seaman’ and ‘seaman’ were classified as different ranks.

A distinct query was used to find all unique instances of the ranks, to retrieve every possible rank type. The results were then placed in a list. An aggregate was used within a for loop to find all instances of unique sailors being at each rank using a match pipeline and using the rank list contents. The documents from each aggregate loop were counted and added to a new list. For example, if the rank ‘Captain’ was first in the rank list, then the count for captain would be in the first position of the count list.

```
for doc in cursor:
    rank_list.append(str(doc))

#searching database for sailor indiviuals at each rank
for i in range(len(rank_list)):
        num = 0
        cursor = db.jah79.aggregate([{"$match": {"mariners.this_ship_capacity": rank_list[i]}},
                                     {"$group": {"_id": "$mariners.name"}}])

        #for the number of records (sailors), add 1 to the counter for that rank
        for doc in cursor:
            num = num + 1

        #add the number to the count list
        count_list.append(num)
```

A considerable amount of missing data and invalid data was found upon early versions of the code in this function. To deal with this issue a while loop was used to loop the results of both lists and remove the rank and its corresponding count if either it was invalid data (e.g. $ or blk) or less than 15. Items that are below 15, before they are removed, have their count added to an added item in the rank list title 'other'. This rank placement holds all small counts in one 'rank' to clear up the pie chart and provides for easier reading of the results.

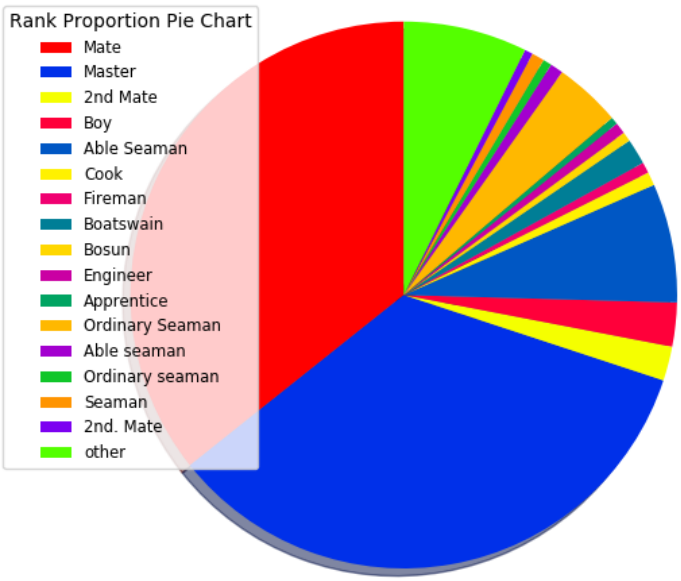
```
while True:
    if int(count_list[t]) < 15:
        other = other + count_list[t]
        del count_list[t]
        del rank_list[t]
    elif str(rank_list[t]) == "$" or str(rank_list[t]) == "blk":
        unknown = unknown + count_list[t]
        del count_list[t]
        del rank_list[t]
    else:
        t = t + 1

    if t == len(count_list):
        break

count_list.append(other)
rank_list.append("other")
```

The plot was created using functions from the matplotlib library. An option was added at the end of the function to allow the user to see exact statistics of the pie chart should they wish to see them. The pie chart colours are unique to make the chart results identifiable, but some colours are very similar and can make it difficult to differentiate the objects.

To reduce the number of repeated ranks (e.g. 'Ordinary seaman' and 'Ordinary Seaman', '2nd Mate' and '2nd. Mate') a split could be used to split the rank words and when the count is executed, it would then include the records that include those two words regardless of other additions/variations. However, there are records with multiple ranks within the same field. Therefore, if a sailor with ranks of '2nd engineer & mate' was searched, it would return as a '2nd mate' result too, a false positive. With this pie chart, I have tried to find a balance in the data used but it is difficult due to the state, consistency and non-atomicity of the data.



The user has the option afterwards to view the statistics that the graph was based on. If this is selected, the following print is displayed on the terminal.

```
RANK - NUMBER AT RANK
=====================
Mate - 1391
Master - 1334
2nd Mate - 79
Boy - 101
Able Seaman - 274
Cook - 31
Fireman - 25
Boatswain - 59
Bosun - 22
Engineer - 26
Apprentice - 17
Ordinary Seaman - 154
Able seaman - 30
Ordinary seaman - 20
Seaman - 29
2nd. Mate - 19
other - 285
```

Visualisations of promotion track comparison

I struggled to get two lots of sailors' data extracted and added to the final graph. As a simple fix, to allow comparison of sailors, at the end of the function the user is asked if they wish to re-run the function with another sailor. As the files save with the sailor's name, both files will be saved without being overwritten.

The function for this task requires user input of a sailors' information. A large aggregate is used with various pipelines to first match the text in the name field with the names the user input, then to unwind the mariners records. The records at this point are sorted with a positive sort on the mariners.this_ship_joining_date field which is the field which best fits the purpose of dating the ranks of a sailor. A match then follows to retrieve only records that have the same year of birth as the user entered. The final pipeline in the aggregate groups the rank and the joining ship date to form the id and what will be returned.

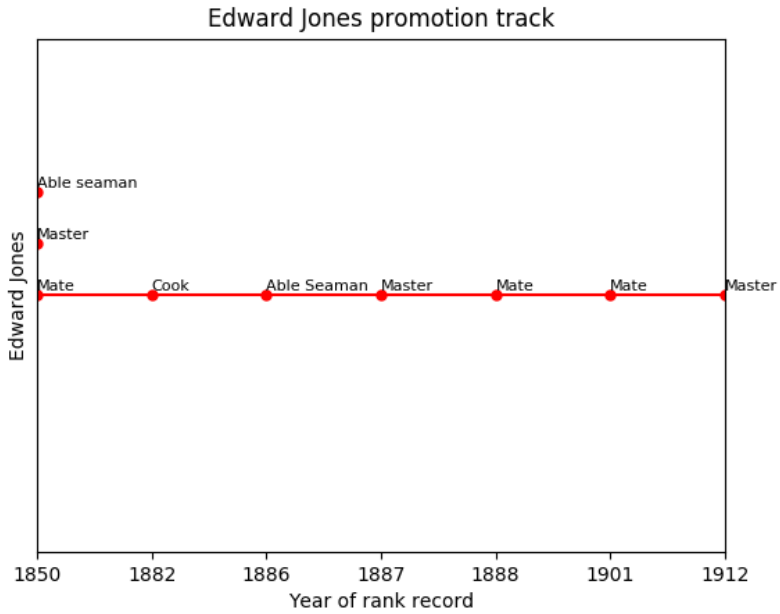
```
cursor = db.jah79.aggregate([{"$match": {"$text": {"$search": str(searchkey)}}},
                             {"$unwind": "$mariners"},
                             {"$sort": {"mariners.this_ship_joining_date": 1}},
                             {"$match": {"mariners.year_of_birth": int(s_year)}},
                             {"$group": {"_id": {"rank": "$mariners.this_ship_capacity",
                                                 "date": "$mariners.this_ship_joining_date"}}}
                             ])
```

The rank and date are extracted using a similar for loop method as those used in previous functions to attain the same results. Invalid data is found and removed and any dates that don't begin with the number 1 are also removed as they would be incorrect also. The date list is converted into an array using '*np.asarray(dates)*'. The dates and ranks are rearranged so that the earliest date appears at the top of the list with the correct corresponding rank appearing in the same position but in the rank list. This was done by repeatedly finding the smallest value in the array and adding it to a new list and adding its corresponding rank to a new list too.

The graph was plot using the matplotlib library again. However, after the initial plot, it appeared that three of the test plots were on the same date and overlapped one another making it difficult to read the text or see the plot. To solve this issue, a for loop is used to check the count of each date before plotting and adds a small value to the y axis for each extra rank on that date. The code for this is shown below:

```
for k in range(len(newranks)):
    sailor_pos = 0.5
    for e in range(k):
        if newdates[k] == newdates[k-(e+1)]:
            sailor_pos = sailor_pos + 0.1
    sailor_position.append(sailor_pos)
```

I was unsure on how to create a timeline and searching online lead to only dead ends. As a compromise I created a scatter graph with the 'x' value being the date and the 'y' axis being a constant 0.5 throughout the records unless edited due to repeated dates. The 'y' axis labels were removed and replaced with the sailor's name. A line was added at 0.5 to create the timeline itself. The end result works as planned, shows correct results and is easily readable.



Similarly to the pie chart function, once the plot has been saved, the user is asked if they wish to see the statistics that went into the making of the graph.

```
Year - Rank
============
1850 - Mate
1850 - Master
1850 - Able seaman
1882 - Cook
1886 - Able Seaman
1887 - Master
1888 - Mate
1901 - Mate
1912 - Master
(csm6720env) central:~ $
```

Extracting ship ports

To list all the ports each ship has visited, an aggregate was first used to extract every unique vessel name and port of registry. Using the results and a similar dissection method as used in previous function, the text is dissected, and the results stored in two lists. The same thing is executed but with this_ship_joining_ports instead of port of registry. At this stage we have a total of 4 lists, 2 holding ports and 2 holding ship names. A matrix is created and the data from the first ship list and first port list are added to the first two columns. The port names stored in the second list have multiple port names per item in the list. A split is used to retrieve all the port names in that position and a large section of a mixture of for loops and if statements are used to add the new ports from to their correct ship name. A port will not be added if the port name is already in that ships row and if a ports ship doesn't already exist then a new ship is added at the bottom of the matrix.

```
matrix = [[0 for x in range(6)] for y in range(len(ship_names))]
for n in range(len(ship_names)):
    matrix[n][0] = ship_names[n]
    matrix[n][1] = registry_ports[n]

for p in range(len(joining_ports)):
    for q in range(len(joining_ports)):
        if matrix[q][0] == joining_ships[p]:
            collection = joining_ports[p]
            portnames = collection.split(", ")
            for r in range(len(portnames)):
                for s in range(9):
                    if matrix[q][s+1] == portnames[r]:
                        break
                    elif matrix[q][s+1] == "0" or matrix[n][s+1] == 0:
                        matrix[q][s+1] = portnames[r]
                        break
            break
```

The results:

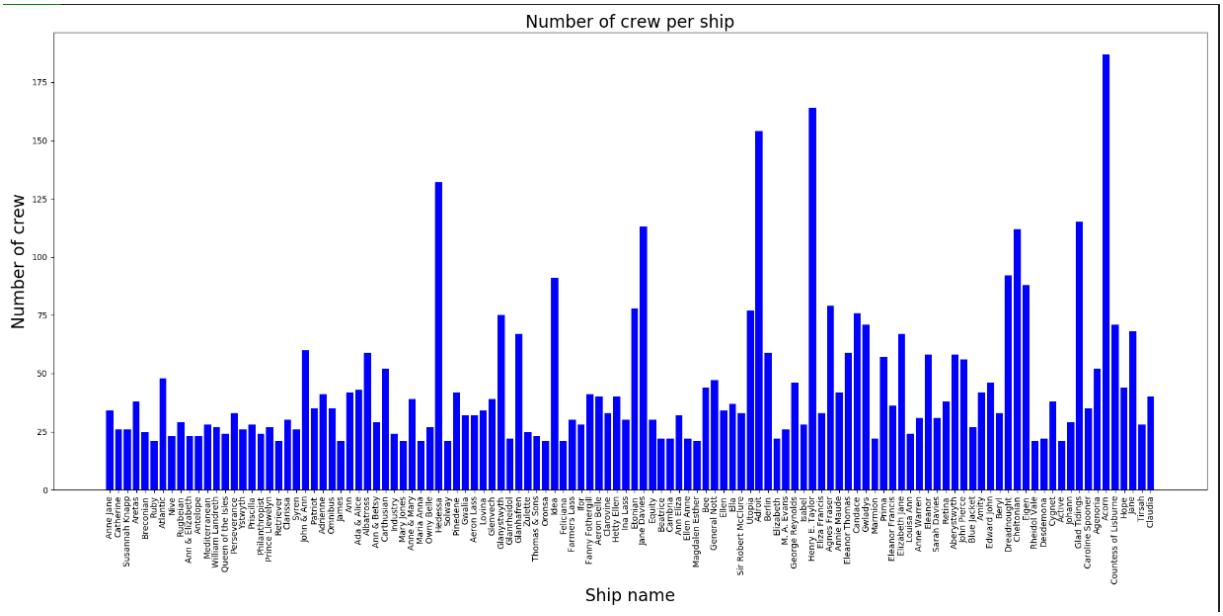
```
'Ship Name (1)// Ports visited (2-6)'
'================================='
[['one, ', 'Aberystwyth', "'blk'", 0, 0, ''],
 [' Mary Jane', 'Aberystwyth', "'Bangor'", 0, 0, "'Cardiff'"],
 ['Acorn', 'Aberystwyth', "'London'", 0, 0, "'Dungarvan'"],
 ['Active', 'Aberystwyth', "'Continued'", 0, 0, 0],
 ['Adela S Hills', 'Aberystwyth', "'Liverpool'", 0, 0, 0],
 ['Adela S. Hills', 'Aberystwyth', "'Cardiff'", 0, 0, 0],
 ['Adoram', 'Aberystwyth', "'blk'", 0, 0, 0],
 ['Adroit', 'Aberystwyth', "'Runcorn'", 0, 0, "'Barmouth'"],
 ['Advance', 'Aberystwyth', "'Liverpool'", 0, 0, 0],
 ['Aeron Belle', 'Aberystwyth', "'Cork'", 0, 0, 0],
 ['Aeron Lass', 'Aberystwyth', "'Aberaeron'", 0, 0, "'Liverpool'"],
 ['Aeron Maid', 'Aberystwyth', "'Liverpool'", 0, 0, 0],
 ['Aeron Vale', 'Aberystwyth  ', "'Liverpool'", 0, 0, 0],
 ['Agnes Cairnes', 'Aberystwyth', "'Poole'", 0, 0, 0],
```

Some missing data found its way into the set ('blk') but the biggest issue here appears to be that some port names leave gaps between the last port position and the port position the are at. I attempted to fix this issue but could not find a working solution.

Visualisation of ships' crew numbers

The assumption was made that the customer means to view the total crew count of a ship over all ports and records. An aggregate is used to return the unique vessel name and its sailors. The text from both those fields is extracted following the previously used method of looping through the characters of the document as a string. There are multiple sailor names in each list index at this point and so the split function is used again to separate these names, count them and add the count value to a new list.

For loops are used to remove the ship duplicates. If a ship with the same name is the same as the ship name currently being checked, then the counter for the ship is added to the counter for the ship currently being checked. The old ship name then has its value reduced to 0. In the following section of code, only ships and their count are added to new lists if the count is above 0. This method removes all duplicates of the ships while retaining the total count of crew for that ship. Items with a count below 20 are moved as there are too many items to display on a graph while keeping the graph easy tor read. The resulting bar chart is clear, the 'x' axis text is small due to the amount of ships, but it is readable and can be zoomed in on.



Added features

A few small features were added, a description of these are listed below:

Menu function

A function was added to produce a menu to allow the user to navigate through the many functions with ease completing the tasks they wish to complete. The menu allow s for the user to select whichever function they require by selecting the number key that matches that function on the menu screen. After each function, the program returns to this menu for the user to select another function should they wish to. 'Esc' can be used to close the program.

Return all records function

Added a second part to the 'retrieve sailor records' section which allows for the user to save all unique records to a text file in case of back up or need of a paper copy.

Options for user throughout the program

Throughout the program I had notifications for the user notifying them about when files were saved but in some cases I also added the option for the user to rerun the function there-and-then. This would hopefully save the user time if they needed to reuse the same function repeatedly.

Results

The results of all the functions return or save what they are meant to. The initial 'list_sailors' function successfully creates a document with the list of sailor names. As displayed in the methodology.

The function which retrieves all the records on a specific sailor retrieves all the records but does not present them in a presentable, easy-to-read, manner. The rank proportions function shows a well displayed pie chart that is easily read and coloured to identify each piece. On the other hand, the pie chart contains multiple of the same rank due to the slight spelling difference with some using upper case and others using lower case. I attempted to solve this issue by using '$toLower' in the retrieval aggregation but it failed.

The function which produces a promotion track timeline works as intended but is missing a key feature which I attempted to implement but failed. That key feature is the ability to add another sailor onto the same timeline and see how their ranks compare. That being said, the user is shown an option to re-run the function and produce another plot for another sailor to compare.

Most of the code used throughout this program is fairly basic and the more complex sections of code are often reused through the many functions. Therefore, I believe that the solutions aren't necessarily technical, but all the functions complete the tasks they were created to complete. Some of the more technical areas of the program are likely areas such as the numerous for loops used within one another to create and add data to a matrix correctly. Another example of more technical code could be the use of the lengthy aggregation in the function that retrieves all the records of a specific sailor.

As seen from the graphs displayed in the methodology section, the graphs are clear and well presented to some extent.

Conclusion

I struggled with this project as I am fairly new to mongdb and have limited knowledge on the language. Among balancing this with other assignments, it proved challenging.

That being said, I completed most of the tasks although not in the most efficient way possible. I believe the state of the database made it considerably harder to complete the tasks as there were so many errors, misspelling or empty data in fields. One of the biggest issues was trying to retrieve exact words or parts of a record from the result of an aggregate. Given more time I think I could have made the program more efficient by converting frequently used for loop methods and sections of code into reusable functions. However, each function in the program required slightly different lines of code throughout the similar for loops and so making the for loops a separate function to dissect the returned record text may not work. Most functions don't accept parameter values as it seemed easier/better for this scenario to retrieve the values that would have been function parameters from within the function instead.

I'm sure there are easier solutions and that in some cases I made the problems more complicated for myself, but I take this experience as an experience to learn from. I have learned a lot about mongodb, pymongo, matplotlib and improved python skills and knowledge while working on this assignment. These are useful developed skills that will likely put to use in the future.

This is not my proudest work, I repeated a lot of code, lacked efficiency in some parts. However, I believe this was down to the difficulty provided by the messy dataset and my limited knowledge. Based on how much I personally found this assignment difficult, I am happy with the results and also slightly disappointed that I could not get the final function to work.

I tried accounting for missing data by using extra fields, (such as using age with year_of_birth in case one of either was missing) but due to the amount of misspelling and variation of words, this proved unreliable in itself. I didn't know which options were best, but I focused on getting the tasks done.

Should this program be remade or worked upon in the future, I believe further more-thorough checks could be implemented in each function which would lower the amount of accuracy damage of results afflicted by the inconsistencies and duplications of data.

As I mentioned at the beginning of the report, validation and verification techniques should be used when a user is entering a new record into the database to lower the amount of misspellings and missing data.

Bibliography

Chen, Q., Zobel, J. and Verspoor, K. (2019). *Duplicates, redundancies and inconsistencies in the primary nucleotide databases: a descriptive study*. [online] PMC. Available at: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5225397/ [Accessed 9 May 2019].