# Intelligent Recommendation Service

Implementation Report

By Brian Davis
1-22-2021

# Table of Contents

# Introduction

With the growing emergence of online content as a service, there is need for the implementation of an Intelligent Recommendation Service, which will allow the company to increase their revenue, by being able to recommend their products to customers using gained knowledge regarding their interests. The program outlined in this report aims to meet the goals and exceed client expectations with capability to enable the comparison of song features through a wide range of similarity metrics and the ability to provide recommendations based on a target entry. The system does this through successfully reading the file, appropriate assignment through suitable data structures and implementation of suitable libraries, alongside key comparison of the suitable features collected from the provided data.

# Problem Analysis

To create the solution, the problem needs to be broken down into steps. The steps needed for this implementation are as follows.

1. Reading of the music dataset file
2. Searching artists and songs
3. Computing Similarity between artists / music tracks by ID
4. Generating recommendations to a user based on their target ID input

To accomplish these steps, it is important to analyse the problem before deciding on the approach to take. There are also a variety of other things to consider when thinking about the best way to approach the problem. One of the problems involved in a Recommendation Service is new users. It's the aim of the service to provide something that draws in new users, but the problem of recommending songs to a new user is that their tastes in music are not existent on the system. To understand the taste of the user, it is important for the program to be subjective and not recommend many things to a user until there is more data to make use of, so the algorithm can be accurate in its assumptions. It can also be a good idea to recommend songs to a user to see what the response would be.

The steps defined above showcase the overall direction of the implementation of the system, and the stages that must be considered when planning the implementation documents. There needs to be an avenue to allow the user to be able to search by name or closest match. The metrics to be created for the similarity scoring need to be tested for suitability of purpose, as not all metrics are useful for this task, but having some variety is good to give a potential user some options. Generating recommendations for the user also depends on their target, and there are a few different types of recommendation styles that can be used. In this case, the program uses Content Based Filtering, which takes a target from the user and generates recommendations based on this target in real time. This style is referred to as a 'Cold-Start' approach where we have no previous user data to work with.

# Solution Requirements

This section outlines the characteristics of the solution, and how these characteristics enable the program to meet the needs of the stakeholders and the business.

## Functional Requirements

When considering the functional requirements, there are a few aspects that are key for this program. The most important of these is the **user requirements**, which can be showcased in a use case diagram (see appendix). This style of diagram helps to show what the program does and how this meets the requirements of the user. This highlights what the program does and how it meets what the user would be expecting when they are using the program.

Another important requirement to consider is the **system requirements**. For this program, the system requirements are small. The main requirement to run the program is a system that can access Python and run a python notebook in the environment of their choice. This could be through Jupyter Notebook or Visual Studio Code using the python add-on, as the system uses a simple UI for a simple program with effective ease of use, including some modular flexibility. An important thing to note here is that the modules that need to be imported must remain in the same folder as the solution, as not to break functionality.

## Non-functional Requirements

The non-functional requirements aim to define the system behaviour. This section discusses these requirements and how they must be met when creating the solution.

The non-functional requirements have been listed in this section. The first is **Usability**, which refers to how easy the program is to use for an end user. The program will use a simple UI, which will require input from the user in the form of text entry. The use of this simple UI for the program input makes the solution efficient, intuitive and maintains a low perceived workload. There are plenty of text outputs for the user so that they understand and can easily follow along and input the relevant response to successfully proceed within the program. Any errors are relayed back to the user in an understandable manner.

Continuing, the next requirement is **Supportability**, which refers to how easy it is to update the codebase when required. The system will make use of Object-Oriented Programming principles with the aim to make the codebase as compact as possible. Making use of OOP allows for the program to eliminate need for unnecessary code duplication in places by making use of Inheritance to share methods from one class to another. Good use of code structure that follows coding standards and contains useful comments will allow a future developer to be able to refresh and update the code when and if necessary.

Another consideration is **Appropriateness**, which is the suitability of the program for its intended purpose. This requirement is simply a measure of how well the program meets its intended purpose, which is to recommend songs or artists to a user based on what they already like. Successful implementation of the program will allow it to meet this requirement without any issue.

The program also needs to be **Reliable**. This involves the effective use of exception handling, which makes sure that the program doesn't experience any crashes if the user inputs a value that would normally create an error, such as when a data frame is called when it is not yet instantiated. The program will therefore make effective use of built-in exception handling to catch all exceptions in the program, and instead of crashing, will print a message to the user, and then re-run the section of the program that was interrupted due to an error if appropriate.

The last non-functional requirement is **Performance**. The program runs through a single execution that calls multiple modules through various sections of the UI. This makes performance of the program fast; the lightweight nature of the implementation will allow the program to run fast and snappy when being used. While there is no need for the program to run fast, the nature of the implementation makes a fast-running program easy to accomplish. The slowest aspect of the program is when the user enters their target, and the program calculates the metric of the target against the library to find recommendations.

# Implementation of Solution

This section outlines the steps taken to accomplish program implementation. More thorough discussion regarding the execution of the program can be found in later sections.

The implementation of the program was an incremental process, which involved working in steps to complete sections of the codebase and returning if any issues came up on previously completed sections. Use of GitHub for saving work and creating issues helped to gauge progress. To accomplish the file loading, the Python library Pandas was used within a file loader class. Using this library meant that all encoding and splitting of data was done automatically using a comma as the delimiter.

Three Classes are used to split up the data as Artist, Song and Extras, where the artist name is included in the Artist class, the song features used for comparison alongside the song name are within the Song class, and the extra features of the data that are not used, except the features Explicit and Instrumentalness, are within the Extras class. A Track class uses Multiple Inheritance to inherit all the keyword arguments and methods from these three classes and uses them to create a combined class for the entire dataset. Justification for this use of Inheritance was to help split the data where appropriate but also keep it together at the same time as a combined class. As the program takes an ID number as the input, if the user doesn't know the ID, then the search function will help them to find it, to pull the information for comparison or recommendation.

A method was created within the Track class to return a data frame to be used later for calculations. Functionality to present the number of successful matches within artist and song searches allows user feedback regarding the accuracy of their search, and then presents them with a box asking them if they want to view the results. This was done so that when a user conducts a search, the program doesn't simply throw all the results at the user without some sort of numeric feedback beforehand. The user can reject the search results and move on in the program if they wish. Searching can be done infinitely until the user has had enough and wishes to proceed, this adds flexibility and some modularity to the solution.

Creation of the metrics was done through method creation and use of the NumPy and SciPy libraries to extract the metric methods needed, which were then called using dot notation. The overall functionality of the similarities includes error checking to avoid incorrect input. When the same numerical ID was entered, the program will output 1 for comparison. A default metric was added so that incorrect entries will result in this default being used, which improves upon the implementation used previously.

The recommendation class was created using inherited properties from the Similarity metric class which houses the metrics to be used. A choice method was also created to define the metric to be used, as some of the metrics require the sorted results to be reversed and some don't, so this needs to be specified in the code to avoid presenting false results to the user. To keep the data to a similar scale, a scaler function was considered which normalised all the values within the data before transformation occurs to turn the data into a vector. Calculations on a data frame took a long time, so the data was transformed into a vector using numpy which turned out to be much faster in presenting results. Transformation was explored as some of the feature values such as loudness and tempo are much larger than others. This scaler was removed after findings in the evaluation stages.

There was the issue of the program returning the same song / artist that are being used as the target, which was corrected by removing the target from the library before any recommendations can be calculated, and skipping the artist being printed if the name already exists as the target artist. The algorithm is chosen by the user and then the program takes the target and loops through the library, using this metric on the target v every individual item within the vector before returning a set of the n closest results to the target, where the n value is chosen by the user as a multiple of 5, with 5 being the lowest value accepted. These results are sorted appropriately before they are shown to the user. The K Nearest Neighbor algorithm was used as a sole method due to the inability to use the full classifier for this task. The KNN algorithm was used as the final method as another way to get results that doesn't simply use the metrics created and uses the algorithm from a library. It aims to be not only another avenue for results, but also as an avenue for comparisons of result accuracy in the Evaluation section. KNN also presents results to a user much faster than the created metrics.

# Program Execution

This section discusses the implementation and execution of the program through the main notebook. The section focuses on the choices made and the structural decisions when deciding upon program flow and flexibility.

The program executes through the main notebook by calling the Main class. This class was created with methods that call methods from the other classes within the two imported modules. The program is easy to use with an intuitive UI that guides the user through their selections and presents suitable feedback when incorrect entries are identified. The program will attempt to recover when errors are identified in inputs, but where this wasn't possible, the program will need to be started again by re-running the code block. The use of methods within the classes aims to reduce code duplication at every possible opportunity. Prompts to ask the user if they want to quit after errors is added.

A flow chart is designed for an easy-to-read diagram that showcases the overall flow and flexibility of the program, dependant on the instruction of the user. This flow chart can be found within the appendix. It was important to create a program that closely matched the flow chart, and it is successful in that regard. A System Architectural Diagram is located within the appendix and shows the design for the programs intended functionality. There is also a Class diagram within the appendix which is required to show the relationships between the classes that have been created for the solution.

When creating the main program, it was important to give the user a choice of what they would like to do. For this, markdown guides the user within the notebook, alongside the use of printed output to guide the user within the program. A main function is present, but code blocks are available toward the bottom of the notebook, to allow a modular execution if this direction is preferred. Pseudo code for the main function can also be found in the appendix, alongside pseudo code for all created classes and methods within the solution. Evaluation code can be found towards the bottom of the main notebook.

The relationship between the modules is also of note. The program first uses the load dataset module to get the data, after which the program will then call the methods defined in the similarity module. While there is no real direct relationship between the modules, without the load module, the program can't function as the code will falter without the loaded data. The strongest relationship is with the classes found within the similarity module, which all work in tangent to allow the user to search for artists and the use of metrics to get accurate results printed to them.

# Personal Reflection

This section provides an overview of the main issues found within the program during implementation and outlines the ways that they were corrected where necessary. The section mainly discusses the implementation of the two modules for the program, and the implementation of the main notebook.

## Dataset Loading

Loading the data is the first part of the implementation. Working with the file was seamless whilst making use of Pandas. The use of zip allowed the data to be added to the Track class as a class-based instance that could be instantiated, which essentially made the file a dictionary with additional functionality. Indexing is done automatically by Pandas. IDs are present in searches by the user for songs or artists.

## Similarity Metrics

The program uses 5 similarity metrics to run comparisons on features and generate recommendations. These metrics are Euclidean, Manhattan, Pearson Correlation, Cosine and Jaccard. When working with these metrics, it was increasingly important that rigorous testing was conducted to make sure that there were no issues regarding inputs, outputs, and calculations of the results. The premise of comparisons is a value against another value, whilst recommendations work by taking a target set of values and comparing them to all other values for each other item in the library. A problem that arose in using two features of shape (1, ) is that Pearson Correlation shouldn't be used in this respect. When using correlation for recommendations, no issues were discovered akin to those for comparison.

Euclidean and Manhattan are successful for comparing and recommending items for a user, as these algorithms work by taking the first item from the second and applying some other math to the results to make them differ from one another. Comparisons done with these two metrics shared the same result, but for recommendations, these results were no longer identical in score. Euclidean and Manhattan are accurate across all values, leading to a successful implementation of these metrics. The same can be said for the metrics of Cosine and Jaccard, where the issues found when working with these metrics were small, although Jaccard, Cosine and Pearson are not suitable metrics for comparison, as most of the time the output is 0 or 1, and this is not very informative, due to the way these metrics work algorithmically. Evaluations against KNN showed the created metric methods outputted exact results.

## Generating Recommendations

When deciding on the best approach for generating n recommendations, the first initial thought was to adapt the solution used when the program compares all features from an item against another item. To generate the recommendations, it was decided to take the id number of an item and loop through all other items and do the metric calculation of all features of an item against all other items instead to generate a more accurate recommendation engine.

The use of a data frame to implement this was initially very slow, due to the dimensionality of the data. Instead, transforming the shape of the data into a vector, and then calculating similarity proved to be incredibly fast in comparison, with the slowest result taking 10 seconds using Pearson Correlation. Once these results were compiled and added to a list, they were sorted by their scores and then the

IDs of the ordered results were used to pull the names and print them to the user. The use of the class-based list from file loading proved incredibly useful for this. The loop to print results to the user will then terminate when the number of printed results reaches the value of the n that the user enters.

## Main Function

The aim of the final implementation of the UI was to create an intuitive and user-friendly experience. The structure of this UI had to follow the flow chart exactly and this was successful. Implementation of the UI didn't have any critical issues, as the use of OOP principles made the UI very easy to code within the Main class, making use of methods to avoid code duplication where this was feasible to do so. A while loop is used to allow a user to search until they enter continue. An issue not corrected for the search results is the abundance of results being printed all at once rather than in increments. This is only really an issue when the user searches for a song rather than when they search for an artist, due to the way the song search was implemented, which looks for words in song titles, and entering more than one word would result in a lot of results due to the weakness of the song search implementation. A results box is part of the UI to avoid all results being printed without the user asking to see them, but the issues with the song search functionality remain unresolved.

Input boxes can be prone to spaces being entered. This was avoided by using the strip() command for inputs. This would take all leading and trailing spaces away from any inputs. Capitalisation was used for string-based inputs as the names of features are capitalised within the data frame, so even if the user enters all upper case or all lower case, their input will still be correct unless the spelling is incorrect. This did have to adjusted for artist name searching due to artists such as AC/DC which is fully capitalised. If results are empty, the program will try the entry in full caps before ending the current search.

# Evaluation

To decide which is the best metric in terms of the overall recommendation accuracy, there is need to evaluate the recommendations given by each metric against the others. A dummy set is created with a dummy target to see how each metric decides based on the values of the target, in what order to recommend items. Looking at the results, it is possible to see that all metrics rank the dummy items similarly except for Jaccard, where the accuracy of each other metrics ranking against Jaccard is 10%. This means that the metrics other than Jaccard have a similar accuracy when we define a dummy set of items. This is explored more on a target ID from the dataset, where findings suggest that the most accurate metrics are Cosine and Pearson, which are 60% alike. Euclidean and Manhattan are also similar, but with only a 30% similarity to one another in terms of the ranking results for this target.

Plots help to show the overall mean of the item values and how close to the target mean of values these recommendations are. For Euclidean and Manhattan, the target and recommendations have very similar mean of item values, where the other metrics have a varied mix considering the mean of item values. Overall, there is some confidence to suggest that Euclidean and Manhattan output a similar order to their recommendations for the first 3 or 4 items, whilst Pearson and Cosine output similarly for the first 4 to 6 items on average. This is simply due to the way these algorithms work, which makes sense that they align to each other more than they do to the others when considering ranking of the items based on the target. Euclidean and Manhattan can be easily influenced based on the mean of the other items, whilst Cosine and Pearson don't have this problem. For accuracy, based on algorithmic efficiency, I would go with Cosine or Pearson as the best performing and overall best metrics for the task, as they aren't influenced by mean. Cosine is used as the default metric where possible.

# Conclusion

This report showcases the steps taken to create a final runnable solution for a potential user and contains a brief evaluation of the recommendation metrics' results. The main aims for the project have been fulfilled and the searching feature is included as it is necessary to allow a user to find the target ID that they want to use to get recommendations. Overall, the program has reached a suitable state that all parties can be happy with, and recommendations can now be simulated and presented to a potential user.

# References

Harris, C.R., Millman, K.J., van der Walt, S.J. et al. *Array programming with NumPy*. Nature 585, 357–362 (2020). DOI: 0.1038/s41586-020-2649-2.

Van Rossum, G. (2020). *The Python Library Reference release 3.8.2*. Python Software Foundation.

Schedl, M., Zamani, H., Chen, C.-W., Deldjoo, Y., & Elahi, M. (2018). Current challenges and visions in music recommender systems research. *International Journal of Multimedia Information Retrieval*, *7*(2), 95–116. https://doi.org/10.1007/s13735-018-0154-2

*What is a Flowchart*. (n.d.). Lucidchart. Retrieved 28th November 2020, from https://www.lucidchart.com/pages/what-is-a-flowchart-tutorial

*Guthrie, G. (2020, August 2). Everything you need to know about architectural diagrams (and how to draw one). Cacoo. https://cacoo.com/blog/everything-you-need-to-know-about-architectural-diagrams-and-how-to-draw-one/*

*Functional vs Non-Functional Requirements: The Definitive Guide*. (2019, September 30). QRA Corp. https://qracorp.com/functional-vs-non-functional-requirements/

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E.A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. (2020) ***SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python***. *Nature Methods*, 17(3), 261-272.

John D. Hunter. **Matplotlib: A 2D Graphics Environment**, Computing in Science & Engineering, **9**, 90-95 (2007), DOI:10.1109/MCSE.2007.55

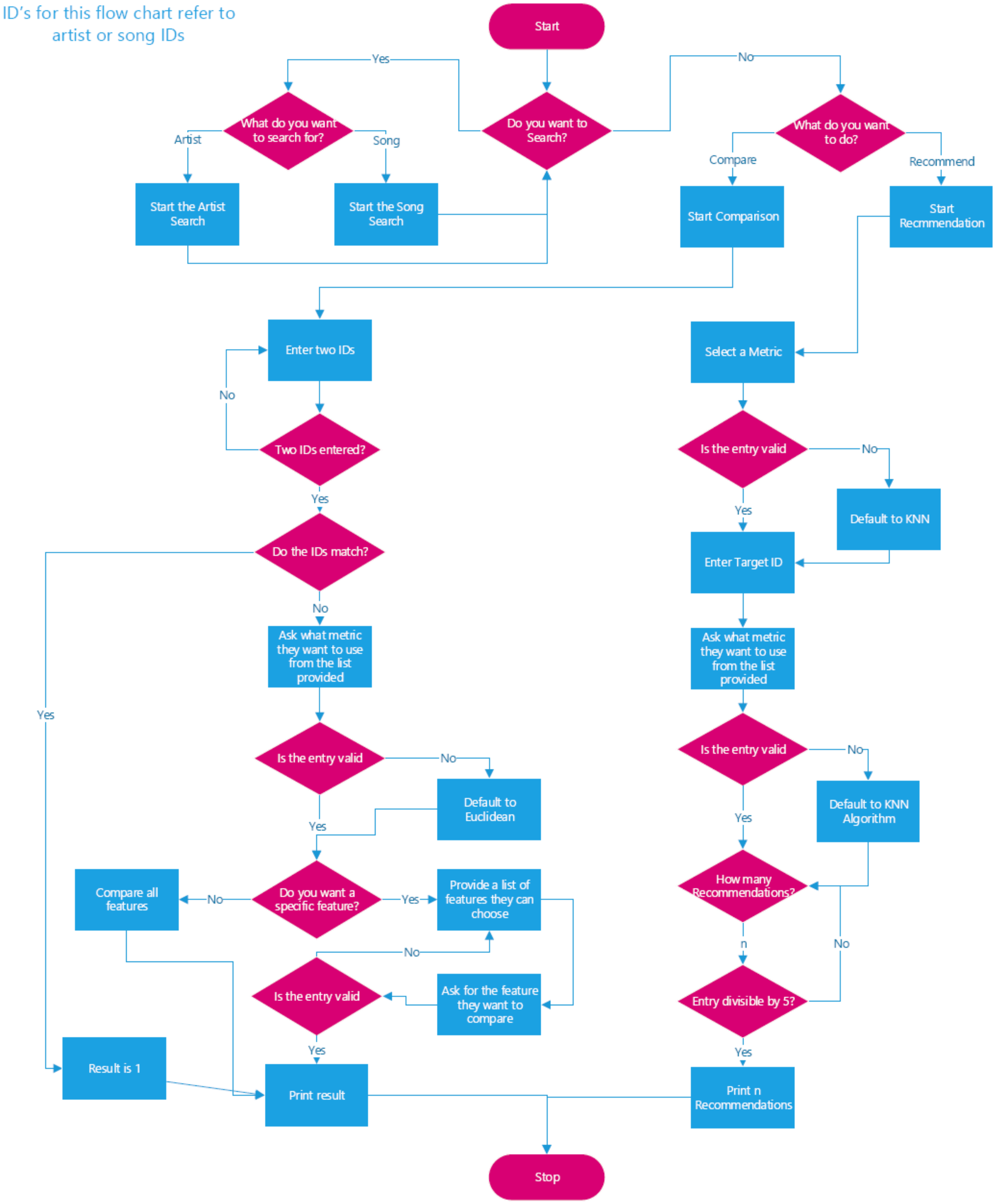
Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É. (2011). ***Scikit-learn: Machine Learning in Python. Journal of Machine Learning Research***, 12(85), 2825–2830.

Python, R. (n.d.). *UML Diagrams – Real Python*. Retrieved 16 January 2021, from https://realpython.com/lessons/uml-diagrams/

# Appendix

## Program Structure Flowchart

The flowchart that shows the overall expected flow of the program is found here, provided below. A small label has been added to the right side of the flow-chart, so that a reader can understand what ID refers to here if they are not sure.



## System Architectural Diagram

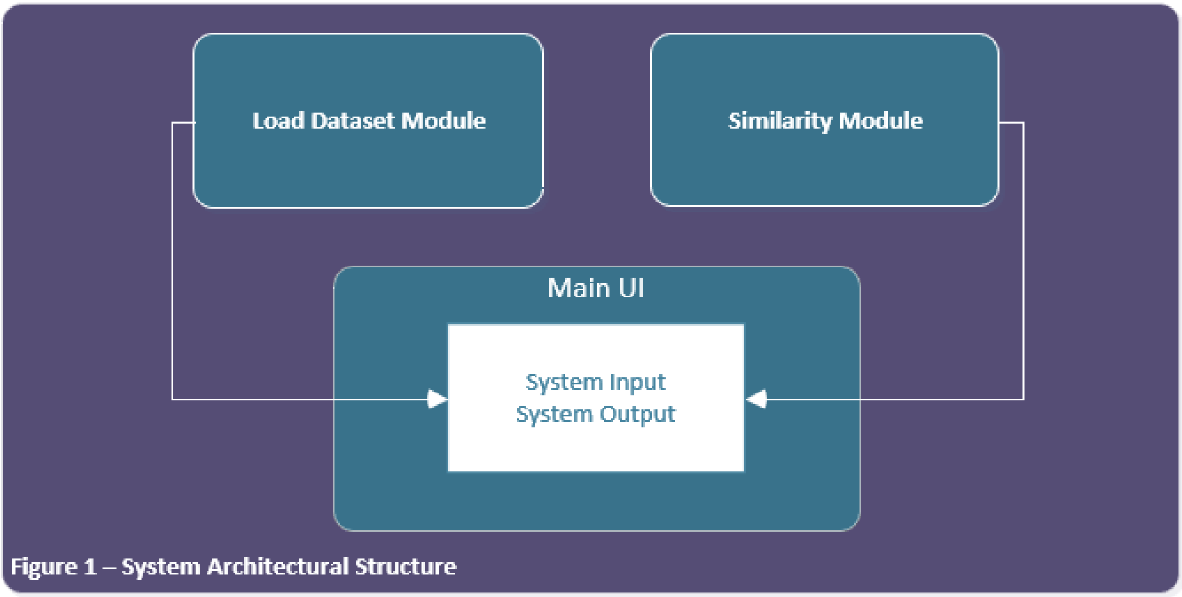


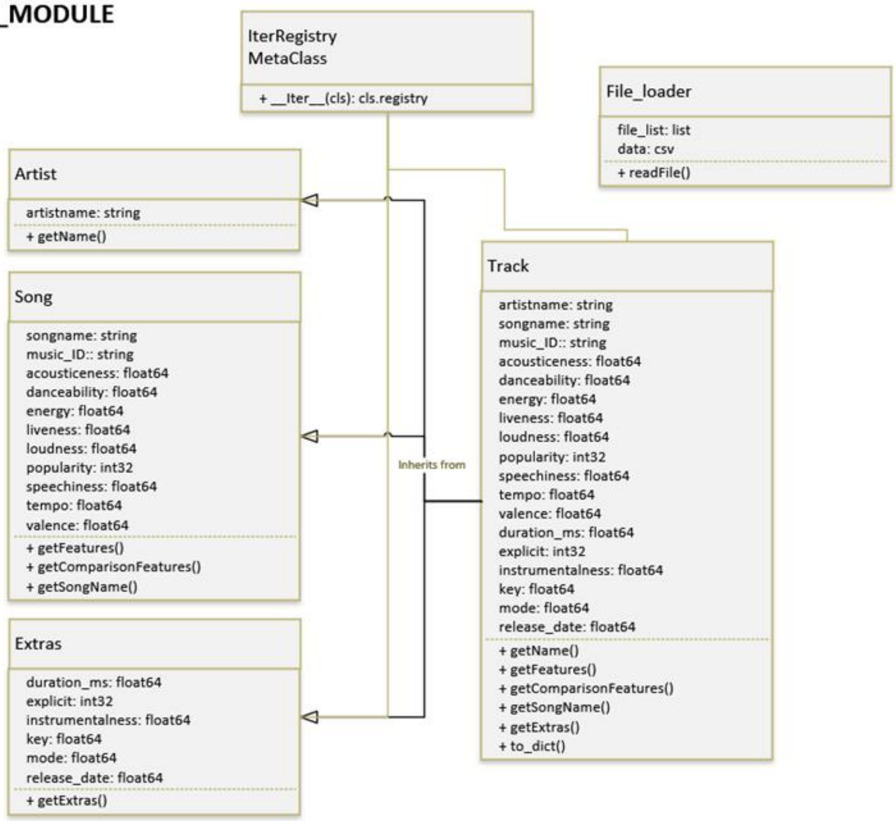
Figure 1 – System Architectural Structure

## Use Case Textual Diagram

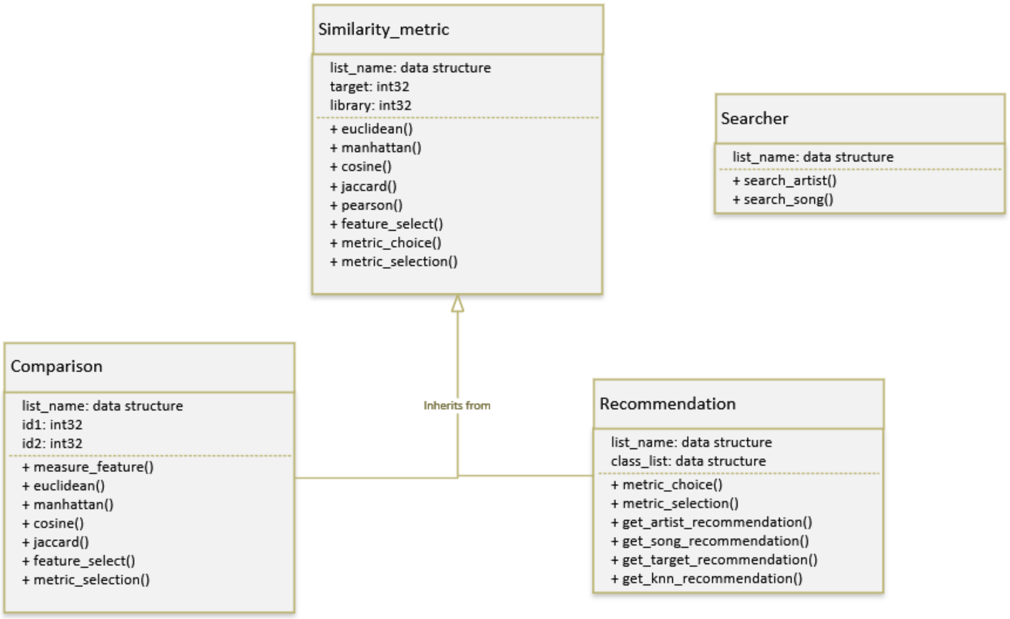
| Actors | Use Cases | Description |
|---|---|---|
| User | Search Artist | Query the Class-Library |
| | Search Song | Query the Class-Library |
| | Enter IDs | IDs to be used for Comparison |
| | Choose Metrics | Metric choice to compare features |
| | Artist v Artist Dictionary Creation | Invoke the method |
| | Rate Music | Convey liking of Music (not included) |
| Admin | Calculate Similarity | Invoke the chosen method |
| | Show Predictions | Recommendation Ranking |
| | Gather Search History Data | Log user activities (search history) |

## Class Diagram

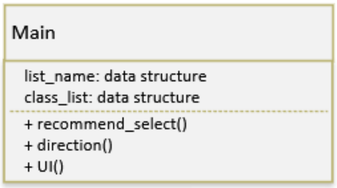
### LOAD_DATASET_MODULE



### SIMILARITY_MODULE

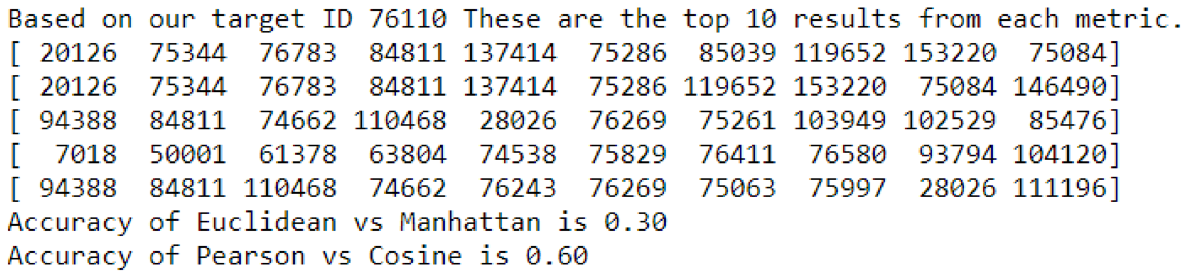


## MAIN

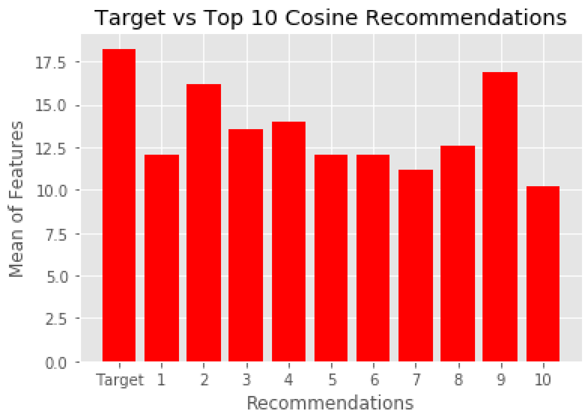


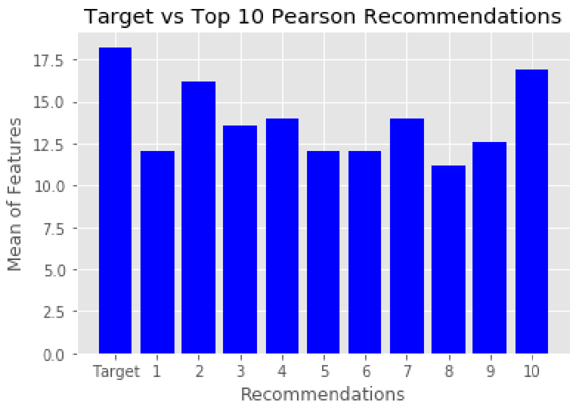
## Evaluation Plots

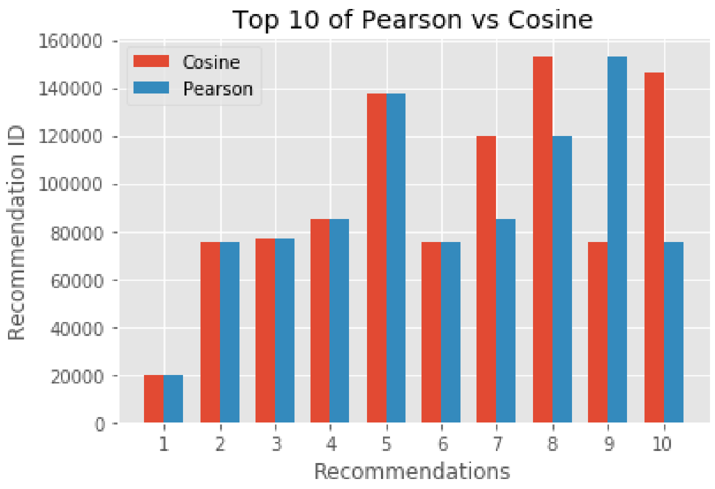
Plots were created using Target ID 76110 without MinMaxScaler implemented

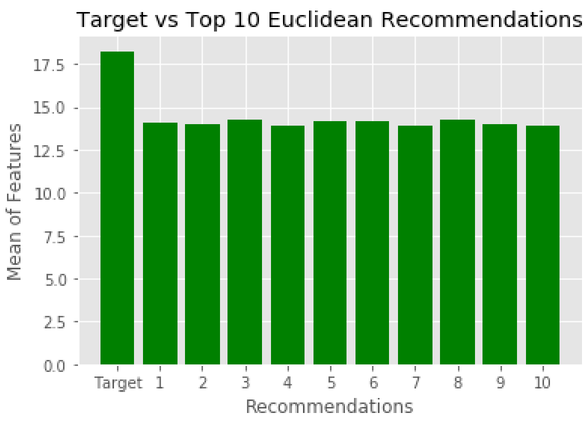


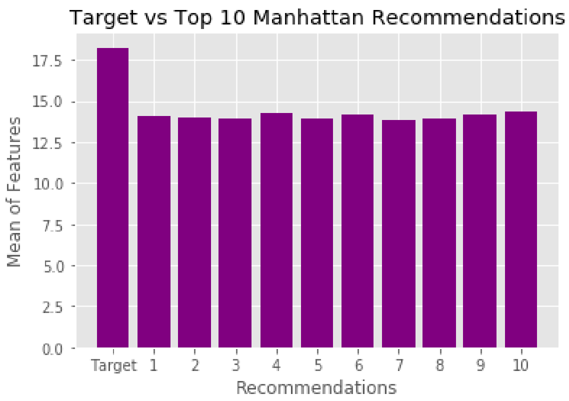
```
Based on our target ID 76110 These are the top 10 results from each metric.
[ 20126  75344  76783  84811 137414  75286  85039 119652 153220  75084]
[ 20126  75344  76783  84811 137414  75286 119652 153220  75084 146490]
[ 94388  84811  74662 110468  28026  76269  75261 103949 102529  85476]
[  7018  50001  61378  63804  74538  75829  76411  76580  93794 104120]
[ 94388  84811 110468  74662  76243  76269  75063  75997  28026 111196]
Accuracy of Euclidean vs Manhattan is 0.30
Accuracy of Pearson vs Cosine is 0.60
```

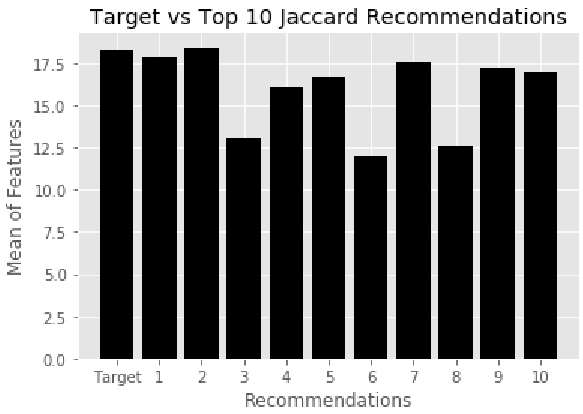


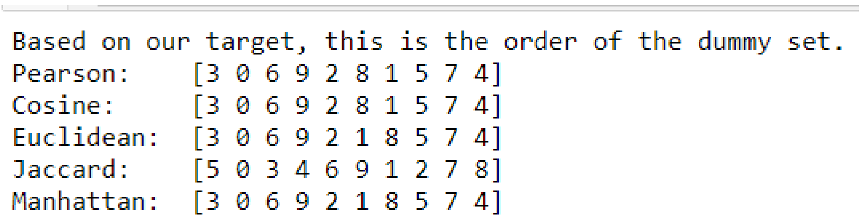




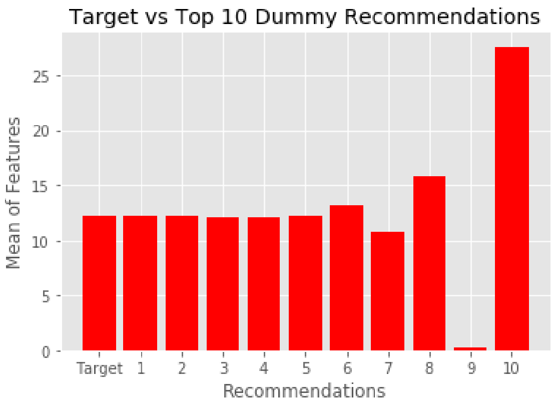








```
Based on our target, this is the order of the dummy set.
Pearson:    [3 0 6 9 2 8 1 5 7 4]
Cosine:     [3 0 6 9 2 8 1 5 7 4]
Euclidean:  [3 0 6 9 2 1 8 5 7 4]
Jaccard:    [5 0 3 4 6 9 1 2 7 8]
Manhattan:  [3 0 6 9 2 1 8 5 7 4]
```



## Pseudocode

### Load dataset module

```
Class Artist
    Initialise with self, artist name and keyword arguments (for inheritance)
        Assign the values
    Define a repr to return string format
    Method getName
        Return the artist name in string format

Class Song
    Initialise with self, song name, music id and the comparison features, keyword arguments to allow IH
        Assign the values
    Define a repr to return string format
    Method getFeatures
        Return the features in a string format
    Method getComparisonFeatures
        Return only the nine comparison features
    Method getSongName
        Return the song name

Class Extras
    Initialise with the extra features, keyword arguments to allow IH
        Assign the values
    Define a repr to return string format
    Method getExtras
        Return the extra features

Class Track which will inherit from Artist, Song and Extras classes
    Initialise a repr to return string format values
        Assign the values
    Define a to dictionary method
        Return a dataframe friendly format that includes the 11 features to be used for calculations
Class File Loader
    Initialise the class, with an empty list and the csv file name
    Method to read the file, take the values and assign them to the list and append that using the track class
        Return the list, so it can be instantiated
```

## Similarity module

**Class Searcher – take a list as an input**

```
Initialise the class with a list as the input
    Assign the values
```

***Method search artist***

```
Take the first name from the user as a string
Take the second name from the user as a string
Take the feature name from the user as a string
Create empty lists for results
For increment I in the range of 1 to the overall length of the artist features dictionary
    If the names entered are in the dictionary at the key for artist names
        Append the feature from the song that was matched with the artist
    If the length of the list result is empty
        Return nothing
    Else the dictionary takes the first name and surname initial as the new key and takes the results
    from the appended list
    Ask if the user wants to see results
    If yes, print the results, otherwise print search complete
```

***Method search song***

```
Take the input from the user of a word that they want to find from the song they are looking for
Strip away any whitespace at the end of the input
Split the values of the input by the space to create a list of words
If the length of the input equals 1, then we must have only one word as out input
    Join the input word back together so it removes it from a list
    For increment I in the range of 1 to the length of the dictionary being searched
        Append this to a new list (not currently done, might not be needed)
Else there are more than 1 word in the entry
    For increment I in the range of 1 to the length of the dictionary being searched
        Capitalise each word in the list
            Increment through and check each word in the list against the dictionary
            Ask if the user wants to see results
            If yes, print the results, otherwise print search complete
            Append this to a new list (not currently done, might not be needed)
```

```
Class Similarity metric – take a list and two ids as input
    Initialise the class, take in a list, and two ids
        Assign the values
    Method Euclidean
        Take the two entries and apply the Euclidean formula, return the result
    Method Manhattan
        Take the two entries and apply the Manhattan formula, return the result
    Method Cosine
        Take the two entries and apply the Cosine formula, return the result
    Method Jaccard
        Take the two entries and apply the Jaccard formula, return the result
    Method Pearson
        Take the two entries and apply the Pearson Correlation formula, return the result
    Method feature select
        Feature select equals a list containing all the names of the features being used (11)
        For each number and feature in the list (using enumerate to create numbers) , starting at 1 (to remove 0)
            Print the number and then the accompanying feature name
                This function will print a list to the user within the program so they know which features
                they can choose from
    Method metric choice
        Ask the user to input a metric selection from the list that will be provided
        Take the column names from the data frame
        If metric choice is 1
            Call Euclidean method with list name, key1 and key2 as arguments
        If metric choice is 2
            Call Cosine method with list name, key1 and key2 as arguments
        If metric choice is 3
            Call Pearson method with list name, key1 and key2 as arguments
        If metric choice is 4
            Call Jaccard method with list name, key1 and key2 as arguments
        If metric choice is 5
            Call Manhattan method with list name, key1 and key2 as arguments
    Method metric selection
        Metric select equals a list containing all the names of the metrics being used (5)
        For each number and metric in the list (using enumerate to create numbers) , starting at 1 (to remove 0)
            Print the number and then the accompanying metric
                This function will print a list to the user within the program so they know which metrics
                they can choose from
```

**Class Comparison –** inherits from similarity metric, takes in a list name and two ids

```
Initialise the class, take in a list, and two ids
    Assign the values
Method measure feature
    Take id numbers if not included in the parenthesis when the module is called
    If the id numbers match, then stop the program and print the result as 1
    Else ask the user for a specific feature they want to compare,
    If the user enters the value of no, or leaves the response empty
        Compare all features defined in the dictionary
        Create two new lists to be able to compare each feature one by one
        Create a list for the column names
        Loop through the range of 0 to 11 excluding 11
            For each value in the list of features in list 1 and list 2
                Use the metric to compute the distance between the values
            Print the result to the user, the program will terminate here
    Else the user entered an invalid feature, or a feature was matched to a key in the dictionary
        If the user entered a valid feature
            Assign features to x and y, Compute the distance metric and return/print the result
```

**Class Recommendation –** inherits from similarity metric, takes in a list and class-based list (used for getting the names)

```
Initialise the class, take in a list and class-based list
    Assign the values
Method metric choice
    Ask the user to input a metric selection from the list that will be provided
        Take the column names from the data frame
        If metric choice is 1, return 1
        If metric choice is 2, return 2
        If metric choice is 3, return 3
        If metric choice is 4, return 4
        If metric choice is 5, return 5
Method get_recommendation (artist, song and target methods are the same except from the outputs)
    Define the needed variables, call the selection method
    Record the metric choice from the inherited method
    Take an input regarding the number of recommendations to present
        If the value is valid, then use the metric they chose before
            Take the target based on the id
            Transform the data frame, drop the target from the library
        If it is not, ask again for the n value
    If the metric choice was 2 or 3 (Cosine or Pearson), we need to reverse the results
    Otherwise return the results in sorted order from smallest score to largest
    For each value in the results
        If the name is the same for the target and the value, skip it
        Print the results until we reach the n selected, then break from the print loop
```

## Main Notebook

```
Class Main – takes a list and a class-based list as the inputs
    Initialise the class, take in a list and class-based list
            Assign the values
    Method recommendations select
        This select equals a list containing all the names of the recommendation types being used (4)
        For each number and recommendation in the list (using enumerate to create numbers) , starting at 1
            Print the number and then the accompanying recommendation type name
                This function will print a list to the user within the program so they know which choices
                they can choose from
    Method direction
        Assign the variables
        Ask the user for an input
            If they choose comparison
                Call the Comparison class’ measure feature method
            If they choose recommendation
                Call the Recommendation class’ method based on their input number
            If the number is wrong, default to option 4 (KNN)
            Print the output to the user
    Method UI
        Ask if the user wants to search
        If they say yes
            Ask them to decide artist or song search
            Call the searcher class’ method for artist or song dependant on input
            Repeat until they want to continue after receiving results
        If they say no
            Call the direction method defined above
            Print the results
        If there are errors, end the program
```

## Requirement Testing Matrix

| Requirement | Use Case | Response |
|---|---|---|
| User can choose artist or song | Enter artist and song | Program proceeds |
| | Enter invalid entry | Program complains and asks if user wants to quit |
| Artist Searching | Does user want to compare?<br>User enters 'yes' | Program continues, starts artist comparison |
| | Does user want to compare?<br>User enters 'no' | Program continues, prompts the user regarding artist searching |
| | User enters a correct name | Program prints search results, prompt for the user |
| | User enters an invalid name | Program prints that there no results |
| | User enters an invalid entry | Program prints a message, starts the metric compare section |
| Song Searching | User wants to search for a song | Song searching begins |
| | User enters an invalid entry | Program restarts |
| | User enters a word | Searching finds songs with the matching word |
| | User enters multiple words | Searching finds songs with any of the entered words<br>**Needs refinement!** |
| | User enters numbers or symbols | Searching returns matches |
| | User enters 'no' | Metric selection begins |
| Metric Selection | User enters a correct number from the list provided | Expected Metric function runs |
| | User enters a character | Program restarts |
| | User enters an invalid number | Program states entry is wrong, defaults to using Cosine |
| Comparison<br>Entering IDs for features | User enters two correct IDs<br>These IDs match | Program complains, asks the user to enter two IDs again as the IDs can't match |
| | User enters two correct IDs<br>These IDs don't match | Program asks for the feature |
| | User enters a character | Program prints an error message and ends, **could be refined** |
| | User enters a number bigger than the size of the list | Program says the feature is wrong, could refine the error message, but functionality is as expected |
| Comparison<br>Feature Input | User enters a valid feature from the list provided | Output the similarity score |
| | User enters an invalid feature, number or otherwise | Program tells the user that the feature doesn't exist |
| | User enters 'no' or nothing | Output all features similarity scores |
| Recommendation<br>Getting N Recommendations | User enters a correct recommendation type | Program proceeds |
| | User enters an incorrect recommendation type | Program errors, and defaults to KNN recommendation |
| | User enters an ID too large | Program errors, restarts the ID selection after choosing metric and n |
| | User enters a correct ID | Program proceeds as expected |
| | Choose n recommendations, user enters a number not a multiple of 5, or 0 | Program errors and tells the user to enter the correct input, restarts the section |
| | User enters correct value for n | Recommendations are presented, program concludes |