Intelligent Recommendation Service

Implementation Report

By Brian Davis
12-4-2020

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. The system does this through successful reading of the file structure, appropriate assignment through suitable data structures, alongside key comparison of 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 to solve this problem are.

1. Loading and parsing of the music dataset file
2. Searching for artists or songs
3. Computing similarity between artists / music tracks by ID

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. Notably, there is no avenue for a user, new or existing, to search for songs within the dictionary. If the user doesn't know what song they want to compare, then there needs to be an avenue to allow them 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 will be useful, but having some variety to choose from is never a bad option either.

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 functional requirement is user requirements. A use case diagram helps to show what the program does and how this meets the requirements for the user. This highlights what the program does and how this meets what the user is expecting to happen. A Use Case Textual Diagram is included below. This showcases some possible future functionality that is not present yet.

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

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 system behaviour. This section discusses these requirements and how they must be met when creating the solution.

Many non-functional requirements were considered when creating the solution, but only three 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 input. The use of this simple UI for the program input makes the solution efficient, intuitive and maintains a low perceived workload. The system will only be required to run a single function for program execution, and modules created earlier will only be called when the function reaches that section of code, depending on user interactions. 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.

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 dictionary 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 function 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, with the slowest aspect being the file reading modules due to the size of the data file. This is of no issue as this should only be done once per loading of the notebook.

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 solution was an incremental process, through the creation of modules and the main function. To accomplish the data parsing, exploration of the file was required. The first task was to read and parse the file, which required encoding at the utf-8 level due to some symbols that were found within names. Another issue was that the columns that contain names have the comma (,) symbol within their values. This was a potential problem due to the nature of the data, the comma found within these values would result in incorrect splitting of the data and would lead to mismatched and incorrect dictionary values. A suitable regex was useful in resolving this, through the detection of a string before a comma, which is then replaced.

To create the dictionaries, it was important to decide what features would be included. There was consideration to include some additional artist features for the artist features dictionary that could be used in comparison, but this was ultimately decided against to create a more concise program for the end user. A considered solution that ended up being included was the ability to find and compare a specific feature of two artists and retrieve the feature for every song that the artist has collaborated on, which is then compiled into a single artist dictionary to allow for comparison through the different metrics implemented.

User searching and the search function are included in final implementation, which helps if the user of the program doesn't know the necessary ID needed to pull the information for comparison. A few extra features were added to artists, but this is purely for informational purposes only.

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 is 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 feedback beforehand. The user also can simply reject the search results and move on in the program. Searching for a song 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 python implementation of the formulas in one-dimensional and multi-dimensional forms. Using the zip keyword allowed the metrics to be used on data that was longer than a (1,) shape, which helped with the overall completion of artist v artist comparisons. Manipulation of the inputs using [ ] allows the zip keyword to also work on one dimensional input.

When creating the artist comparison dictionary, the first thought was which ID would have to be used for the comparison. Initially the ID would be the first name of each artist, but what if the artists share the first name? To fix this problem, the addition of the first initial of the artists' surname was also added as a part of the ID. This fix helped to alleviate this issue, and through testing it was deemed suitable as no artists found in the dictionary appeared to share the same first name and first initial for their surname. If a user enters the same artist multiple times, there is no avenue to stop this, and the value/s will keep being added to the dictionary, since a key in a dictionary must be unique. A way to prevent this behaviour must be considered for further implementations.

The overall functionality of the similarity functions include error checking to avoid the same numerical ID being entered twice, alongside the ability to reset the program if a feature was entered that couldn't be found through a key within the dictionary. Once all metric functions were created and checked for errors, the main function was created. Discussion regarding this can be found in the following section.

Program Execution

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

The program executes through a main notebook which connects all the previously created functions together and calls on them only when required. This is important as it makes the program easy to use and doesn't cause excessive execution, as modules are only executed when the code reaches them. A function is created to aid in the metric selection, and to reduce code duplication. 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. Some variations to the flow chart are present, as there are times when the functionality wasn't always as clear cut to implement in accordance with the intended design. A System Architectural Diagram is located within the appendix and shows the design for the programs intended functionality.

When creating the main program, it was important to give the user a choice of what they would like to do. For this, markdown is used to guide 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. Execution of the program had some initial problems, but these were solved with extra error handling. The ability to raise errors as well was made use of, so that if a certain error occurs, an error can be raised as an avenue to force a program restart. This solution results in less code being required.

To maintain a good program flow, the program will restart sections if an error occurs, rather than simply providing the user with an error message and then halting its process. There are times where it was decided that an error message would initiate an immediate restart of the current function where necessary, there are some times where the user is asked if they want to quit, but the majority of the time the program will restart. The user can initiate the quit sequence from the main menu of the UI. Pseudo code for the main function can also be found in the appendix, alongside pseudo code for all created modules and functions within the solution.

The relationship between the modules is also of note. The program first uses the load dataset module to get the datasets, after which the program will then call the functions 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 defined dictionaries. The strongest relationship is with the functions found within the similarity module, which all work in tangent to allow the user to search for artists and 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

The loading of the dataset is the first part of the implementation. Working with the file was straight forward, aside from some small issues with commas in names which is mentioned above in the Implementation of Solution section. Overall working with the dataset was easy enough and the column names were compiled into a nested dictionary with an indexable key to allow for easy searching for the user. These IDs are also present in searches conducted by the user for songs or artists.

Similarity Metrics

The program uses 5 similarity metrics to run comparisons on features of songs. 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 metric results. The premise of the base comparisons is a single value against another single value, which makes some of these measures inaccurate when doing such a calculation of this shape. The first problem that arises in using two features of shape (1,) is that Pearson Correlation (PC) cannot be used in this respect. This is because a correlation requires at least two x values and two y values to make a prediction of correlation and output an accurate result. As this program only uses one x and one y, the denominator for the formula of PC will always result in zero, leading to a zero-division error which needs to be caught successfully by the program.

Another issue found when using PC to compare a feature from all songs made by a particular artist against another artist is the issue that a lot of the time one artist will have made or collaborated in the creation of more songs than the other. This leads to mismatched shapes of the lists for the PC input. PC is a formula that requires the size of both inputs to be the exact same shape, due to the nature of n for this formula, which leads to more errors that must be successfully caught by the program. The use of NumPy was explored to see if a workaround was possible. The use of NumPy CorrCoef allowed the program to submit a result for 1 x 1 feature comparison, but the result was as expected and not useful in terms of comparing the feature values, although the zero-division error was resolved.

The use of Euclidean and Manhattan for this problem are successful. When comparing values found in a self-created artist v artist dictionary, the resulting numbers for Euclidean and Manhattan are accurate across all values, leading to a successful implementation of these metrics for comparison in the program. 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 is not a suitable metric for this task, as most of the time the output is 0 or 1, and is not very informative, due to what Jaccard is aiming to tell the user about the values they are inputting. To output a result for these metrics, the feature outputs need to be encased in [ ] so that they are recognised as a matrix, which allows these metrics to output accurate results.

Small issues do occur when working with Cosine and Pearson for the feature Popularity when this value is 0 for both features, due to the way the formula for these metrics work, which again resulted in a zero-division error. This was resolved by making it so that when the program reaches the value for I of 5, which would be Popularity when comparing all features, the program will instead continue past it rather than calculate it. This solves the problem, but unfortunately outputs the result for the next feature of Speechiness into the slot for Popularity as well. Pearson uses NumPy now, so this error no longer exists at the time of completion for Pearson.

Another issue encountered with Cosine was a memory error when comparing artist v artist. This was due to the value set for x and y at the time, which required the values of the dictionary to be appended to two separate lists to avoid and solve memory error issues going forward. After implementing this solution, a memory error never occurred in subsequent executions, so has been deemed fixed.

Main Function

The aim of the implementation for the main function / notebook was to create a program that closely matches the flow chart (in the appendix) and provides an all in one UI for user interaction. When creating this function, there were some issues that had to be solved. The first of which was regarding looping for artist and song searching. The plan was to allow the user to infinitely search for artists and songs while the input was 'yes' and would stop only when the user entered something incorrect or said 'no' to the question. This was implemented using a while loop, which for artist functionality checks if the dictionary for artists was a length of 2, and if not, would continue. If the user specified, they could search for an artist without doing artist comparison, in which case the program would continue to present the function for searching, until the user enters 'no' in the follow-up question box.

The same functionality was added to song searching through the same method as the artist searching. Another issue found here was regarding search results, as there are times where a search would present more than 1000 results and sometimes even over 10000, and then this would be printed, which is simply not very user friendly. This was replaced with a results box which would specify how many results were matched, and then ask the user if they want to print the results. If the program says there are over 10000 results, then it is up to the user if they want to print that many or not. This functionality could be improved upon by presenting only the first 20 results each time the user asks to see results, therefore if there are many results, they won't all be printed at the same time, which can flood the output.

A problem encountered through testing was leading and trailing spaces in inputs. It was possible to fix trailing spaces using the rstrip() at the end of each user input, but it was not possible to fix the leading spaces due to the use of capitalisation for inputs and detection. A fix for this would be to remove all elements of capitalisation, but this would break artist and song searching as all the elements of the dictionary are capitalised by word, by default. The fix for leading spaces is to catch errors in the program and relay this to the user as an incorrect input, rather than resolving this through further manipulation of the values within the dictionaries.

Conclusion

This report showcases the steps taken to create a final runnable solution for a potential user. The main aims for the project have been fulfilled and some features such as searching, although not expected of the final product, have been implemented to improve the experience of an end user alongside enhancing program capabilities. There are some features that would make this program even better, but these will be implemented later in a follow-up project. Overall, the program has reached a suitable state that all parties can be happy with. The use of libraries would enhance the capabilities of the program even further by reducing execution time of the dictionaries, although as mentioned in the main report, the amount of times these are expected to be ran are minimal.

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

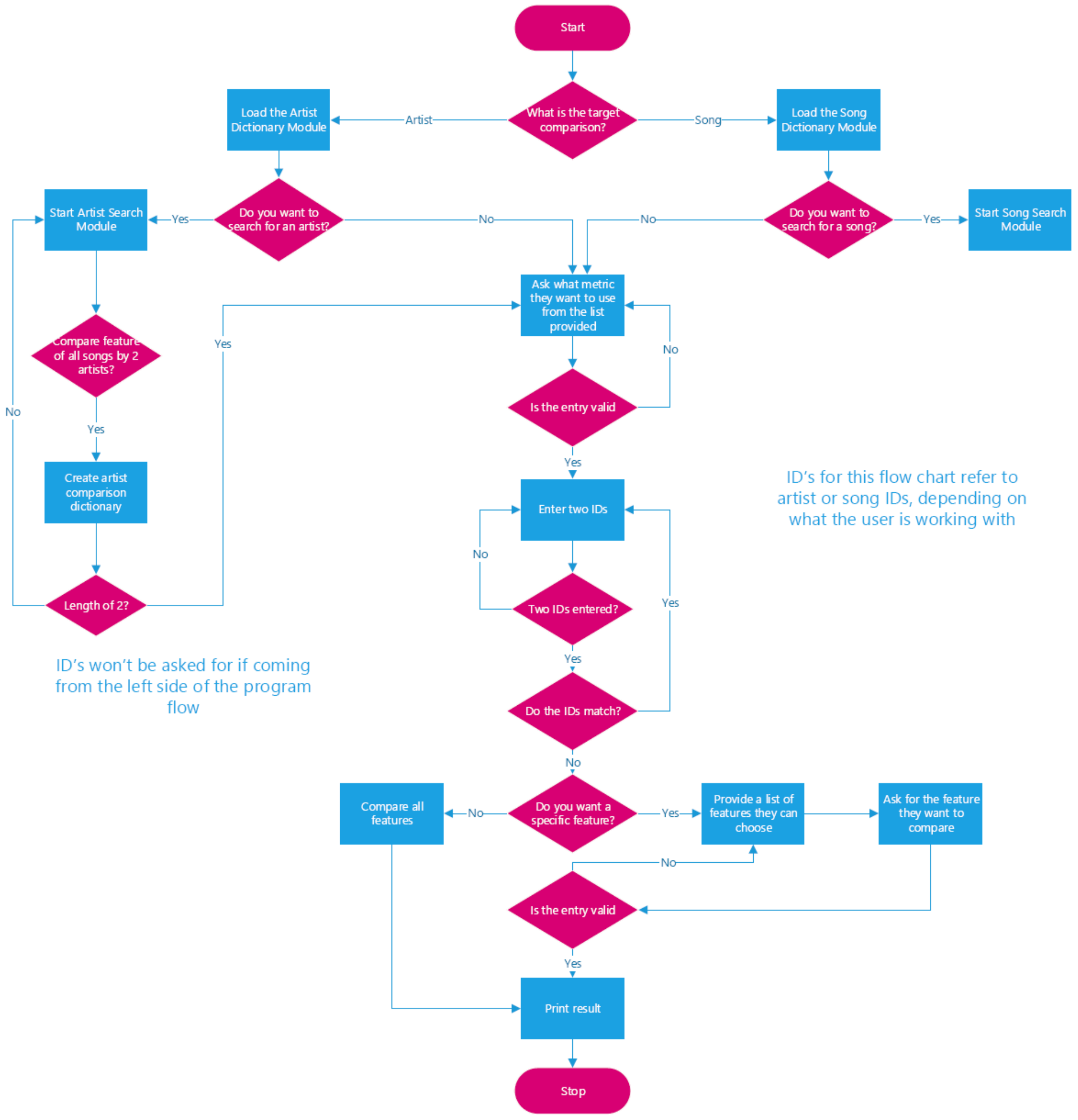
Guthrie, G. (2020, August 2). Everything you need to know about architectural diagrams (and how to draw one). Cacoo. https://cacoo.com/blog/everything-you-need-to-know-about-architectural-diagrams-and-how-to-draw-one/

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

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.



Start
What is the target comparison?
Artist
Song
Load the Artist Dictionary Module
Load the Song Dictionary Module
Do you want to search for an artist?
Do you want to search for a song?
Yes
No
No
Yes
Start Artist Search Module
Start Song Search Module
Compare feature of all songs by 2 artists?
Yes
Create artist comparison dictionary
Length of 2?
No
Yes
Ask what metric they want to use from the list provided
Is the entry valid
No
Yes
Enter two IDs
Two IDs entered?
No
Yes
Do the IDs match?
Yes
No
ID's for this flow chart refer to artist or song IDs, depending on what the user is working with
ID's won't be asked for if coming from the left side of the program flow
Do you want a specific feature?
No
Yes
Compare all features
Provide a list of features they can choose
Ask for the feature they want to compare
Is the entry valid
No
Yes
Print result
Stop

System Architectural Diagram

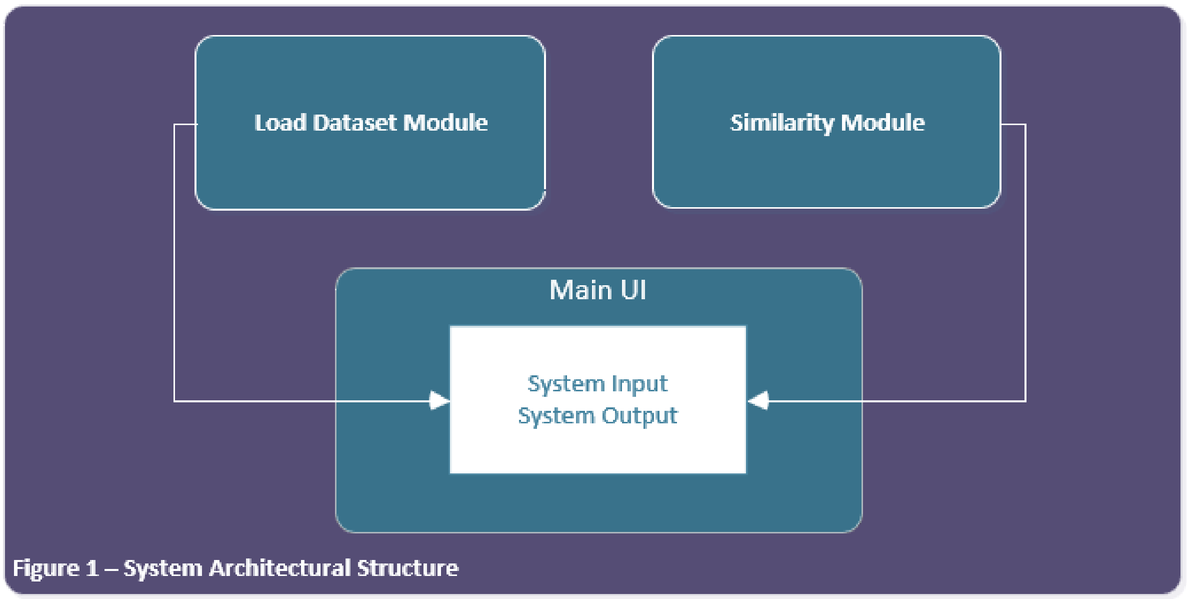


Figure 1 – System Architectural Structure

Pseudocode

Load dataset module

Function artist music / music features

```
With, open the file with the name data.csv, in read mode, with encoding utf8
Create a new dictionary for the result
Create an index that starts at 1
Use the next keyword to skip the headers
For each line in the file
    Using regex, substitute the comma if it is preceded by a string, to a /
    Create a new variable assigned to the column for artist names
    Remove the square brackets and escape characters from those names
    Reassign it to the artist names variable
    Create another empty dictionary d
    Assign each column to a new key in the empty dictionary
    Assign the d dictionary to the result dictionary, using index as the id
    Increment the index number by 1
Return the result dictionary
Close the file
```

Similarity module

Function similarity metric (takes 3 positional arguments of dictionary, id1 and id2)

```
Take id numbers if not included in the parenthesis when the module is called
If the id numbers match, then stop the program and prompt the user
Else ask the user for a specific feature they want to compare, with a small message asking the user to enter Yes if they have created their own artist dictionaries (defined in the function search artist)
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 key names
    Loop through the range of 0 to 9 excluding 9
        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 yes, an invalid feature, or a feature was matched to a key in the dictionary
    If the user entered yes, and the length of the dictionary is 2, this must be a created artist dictionary
        Create empty lists for the x and y variables
            For each value in dictionary id1 and id2
        Append the lists x and y
        Compute the distance metric and return/print the result
    If the user entered a valid feature
        Assign features to x and y
        Compute the distance metric and return/print the result
```

Function search artist (takes 1 positional argument of dictionary name)

```
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 the empty dictionary, and an empty list
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
        Return the dictionary
```

Function search song (takes 1 positional argument of dictionary name)

```
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)
```

Function metric selector (takes no arguments)

```
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
```

Function metric choice (takes 1 positional argument of dictionary name)

```
If the length of the dictionary is 2, then we must be working with an artist comparison
    Ask the user to input a metric selection from the list that will be provided
    Take the key names from the dictionary
    Pass them into new variables key1 and key2
    If metric choice is 1
        Call Euclidean function with dictionary name, key1 and key2 as arguments
    If metric choice is 2
        Call Cosine function with dictionary name, key1 and key2 as arguments
    If metric choice is 3
        Call Pearson function with dictionary name, key1 and key2 as arguments
    If metric choice is 4
        Call Jaccard function with dictionary name, key1 and key2 as arguments
    If metric choice is 5
        Call Manhattan function with dictionary name, key1 and key2 as arguments
This should be repeated for the normal dictionary within the else.
```

This has not been included here as the functionality would be the same in general.

Main Notebook

Main function (takes two dictionaries as arguments – artist and features)

```
    Assign the dictionaries to local function variables
    Print a welcome message
    Print the total of unique artists and songs in dictionaries (make use of both dictionaries here)
    Give the user an option to choose what they want to do (artist or song)
    If the user chooses artist
        Ask what they want to do now, do they want to do artist v artist or just search
        If they want to do artist v artist
            Create a new local dictionary
            Call the artist search function
            Take the result sand add them to the new dictionary
            When the dictionary length is 2, stop the process
            Call the metric select and metric choice functions (this will complete the program)
        If they just want to search
        While the option is yes or artist
            Call the artist search function
        Once the option becomes no or compare
            Call the metric select and metric choice functions (this will complete the program)
    If the user chooses song
        Ask if they want to search for a song
        If the option is yes
            Call the song search function
            Once they are done, ask again if they want to search, continue search until option is no
        If the option is no
            Call the metric select and metric choice functions (this will complete the program)
Program is complete, print a thank you message to the user – this pseudo code aims to mimic the flowchart.
```

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? User enters ‘yes’	Program continues, starts artist comparison
	Does user want to compare? 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 **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, asks the user to retry
Entering IDs for features	User enters two correct IDs These IDs match	Program complains, asks the user to enter two IDs again as the IDs can’t match
	User enters two correct IDs 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
Feature Input	User enters a valid feature from the list provided	Output the similarity score
	User enters ‘artist’ when not working with an artist comparison dictionary	Program says the feature of ‘artist’ doesn’t exist
	User enters an invalid feature, number or otherwise	Program tells the user that the feature doesn’t exist
	User enters ‘artist’ for artist comparison	Output the similarity score
	User enters anything but ‘artist’ in this case	Program throws an error message to the user and ends
	User enters ‘no’ or nothing	Output all features similarity scores