|  |
| --- |
| Music Recommendation Service |
|  |
| Implementation Report |

|  |
| --- |
| By Brian Davis  12-4-2020 |

Table of Contents

[Introduction 2](#_Toc57305975)

[Problem Analysis 2](#_Toc57305976)

[Solution Requirements 2](#_Toc57305977)

[Functional Requirements 2](#_Toc57305978)

[Non-functional Requirements 3](#_Toc57305979)

[Implementation of Solution 3](#_Toc57305980)

[Personal Reflection 4](#_Toc57305981)

[Dataset Loading 4](#_Toc57305982)

[Similarity Metrics 4](#_Toc57305983)

[Program Execution TODO ***can’t be written without the Main()*** 5](#_Toc57305984)

[Program Structure Flowchart 5](#_Toc57305985)

[Module Files Pseudocode 6](#_Toc57305986)

[Load dataset module 6](#_Toc57305987)

[Similarity module 6](#_Toc57305988)

[Appendix 7](#_Toc57305989)

[Functions not defined in the main paper 7](#_Toc57305990)

# Introduction

With the growing emergence of online content as a service, there is need for the implementation of a Music Recommendation Service, which will allow potential customers to find music similar to their own taste and help them discover new content to consume. The Music Recommendation Service 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 MRSP will do 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. Computing similarity between artists
3. Computing similarity between music tracks

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 Music 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 to use in the background for the algorithm to 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 this MRSP, 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 the 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 functionality is not present at the time of submission.

|  |  |  |
| --- | --- | --- |
| 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.

## 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.

Three non-functional requirements were considered when creating the solution. 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 boxes. 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.

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.

The last non-functional requirement is **Performance**. The program runs through a single function that calls multiple modules through various section of the UI. This makes performance of the program fast and 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.

# Implementation of Solution

The implementation of the solution here took place in steps, which included the creation of the modules and the overall creation of the main function which contains the simple UI for the program. To accomplish the data parsing, exploration of the file was required. The first task was to read and parse the file required, which required encoding at the utf-8 level due to some symbols that were found within the artist names and song 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. 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, the first thing was to decide what details would be included in them. The main difference was the inclusion/exclusion of the values of artist names. There was consideration to include some additional artist features for the artist features dictionary, but these were not included to create a more concise program for the end user. A solution chosen for inclusion is 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 dictionary to allow for comparison using whichever metric is chosen. 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. TODO create high level architectural diagram

Once these were complete, the last step was to create the similarity functions for each individual metric. This was done through python implementation of the formulas over one-dimensional and multi-dimensional shapes. 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 added later.

The overall functionality of the similarity functions included error checking to avoid the same ID being entered twice, alongside the ability to stop the program if a feature was entered that couldn’t be found a key within the dictionary. Once all metric function was created and checked for exception handling, the main function was created. Discussion regarding this can be found in the Program Execution section.

# Personal Reflection

This section provides an overview of the main issue found within the program during implementation and outlines the ways that there were correcting where necessary. The section mainly discusses the implementation of the two modules for the program.

## 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 Problem Analysis 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, through functions.

## 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 a result. As we only use one x and one y here, the denominator for the formula of PC will always result in zero, leading to a zero divisible error which needs to be caught successfully by the program.

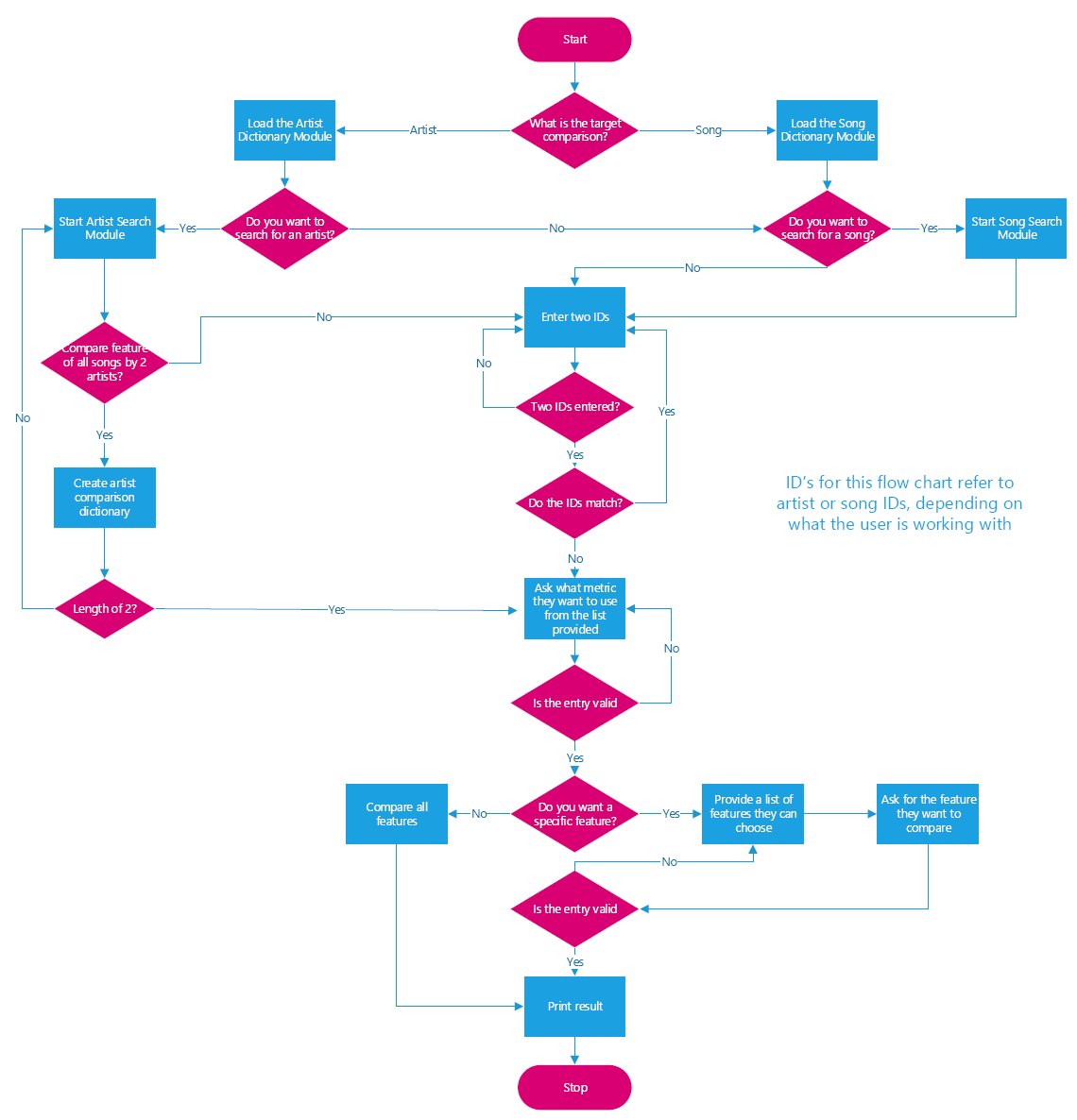
Another issue found when using PC to compare a feature from all songs made by a particular artist against another artist is the issue that a lot of the time one artist will have made or collaborated in the creation of more songs than the other. This leads to mismatched shapes of the lists for the PC input. PC is a formula that requires the shapes 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 divisible error was resolved.

The use of Euclidean and Manhattan for this problem were successful, although most of the time the result of either metric would be the same If the length of the comparison were 1 x 1. When comparing values found in a self-created artist v artist dictionary, the resulting numbers for Euclidean and Manhattan were different, 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. 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 correct results. Small issues do occur when working with Cosine and Jaccard for the feature Popularity when this value is 0 for both features, due to the way the formula for these metrics works. 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, which required the values of the dictionary to be appended to two separate lists to avoid and solve memory error issues going forward. Information regarding formulas for these metrics can be found within the appendix.

# Program Execution TODO ***can’t be written without the Main()***

Discuss the execution process, what went wrong and what went well whilst also discussing the execution process e.g. UI. Include interaction of the UI and the expected flow of this, reference flow chart?

# Program Structure Flowchart

The flowchart that shows the overall expected flow of the program is found in this section, 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. A larger copy of the flow-chart can be found in the attached folder.

# Module Files Pseudocode

This section outlines the pseudocode for each module created. These have been split into two sections, one for each module.

### 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 (take 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

# Appendix

### Functions not defined in the main paper

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

Print the result to the user

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

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

Print the matching results to the user

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

Print the matching results

Append this to a new list (not currently done, might not be needed)