**Your Application Name: SpotiViz**

**Description:**

โปรแกรมค้นหาและแนะนำเพลง

**Data Sources:**

-

**Running the Application:**

* Tkinter : for GUI in python
* Matplotlib: for visualization data .
* Radom: for random.
* Numpy: for advance math and array action.
* Pandas: for data frame and data visualization.
* Networkx: for create network graph
* Sklearn: cosine similarity function for improve shortest path algorithm.

**Design:**

**-**

**Design Patterns Used**

* MVC (model views control) Pattern การใช้ pattern นี้ช่วยให้สามารถแบ่งโปรแกรมออกเป็นส่วนๆได้อย่างชัดเละทำให้จัดการโปรแกรมได้ง่าย .

**Graph Algorithm Used**

* ใช้ shortest-path เพื่อใช้ทำการแนะนำเพลงโดยจะแนะนำโดย ให้เลือกเพลงเป้าหมายจากนั้นจะทำการแนะนำเพลงที่ใกล้เคียงกับเพลงที่เลือกมา 5 เพลง โดยในโปรแกรมจะมีฟังชันแนะนำโดยใช้อัลกอริธิมนี้ 2 ฟังชันมีผลลัพ์เหมือนกันแต่อันนึงจะเป็นแบบบใช้ library sklean มาช่วยทำให้เร็วกว่าแบบที่ไม่ใช้ library

**Be responsive, that is, when it is busy doing some work in the background, the application must not freeze and should always let the user know what it is doing. (ถ้าโปรแกรมกาลังทางาน โปรแกรมจะแจ้งอะไรกับuser)**

โปรแกรมจะมี popup ในการแจ้งเตือนการทำงานต่างๆ ว่าสำเร็จหรือผิดพลาดหรือไม่ มีการจัดวางองค์ประกอบของ UI ที่ผู้ใช้จะรู้ได้ทันทีว่าจะต้องทำอะไร

**Visualize data through 3 types of graphs: distribution graphs (histogram and boxplots), everyday graph, and network graph.**

ในเมนู Chart จะมีให้เลือกแสดงกราฟ Boxplot everyday graph network graph

**Provide informative descriptive statistics and correlation from data.**

มีเมนู report ในการสรุปค่าต่างๆของข้อมูลออกมาเป็นไฟล์ PDF

**• Model a part of the project as a graph problem (reachability, shortest path, 2-coloring, etc.), then use an appropriate graph algorithm to solve it. (ในโปรแกรมใช้ shortest path ในขั้นตอนไหน)**

ในขั้นตอนของการแนะนำเพลงที่ใกล้เคียง

• Modify the provided README.md template to describe what the application does, any dependencies, and how to run it, including links to data sources used by your project. Any work without proper explanation in the README.md file will not be graded.

• Create a requirements.txt file listing all additional libraries used by your application (numpy, pandas, etc).

-

**วิธีใช้โปรแกรมแบบละเอียด, อธิบายการใช้โปรแกรม**

1. เมื่อลงโปรแกรมและทำการติดตั้ง Dependency แล้วให้ทำการรันโปรแกรมที่ main.py
2. เมื่อรันโปรแกรมจะมีแจ้งเตือนให้ผู้ใช้ต้องอัพโหลดไฟล์ ถึงจะสามารถใช้โปรแกรมได้

Graphical user interface, text, application

Description automatically generated

1. หน้าตาโปรแกรมเริ่มต้นเราจะไม่สามารถทำอะไรได้เลยนอกจากอัพโหลดไฟล์ และออก ถ้าเราต้องการที่จะใช้งานเราจำเป็นต้อัพโหลดไฟล์ CSV

Graphical user interface, application

Description automatically generated

Graphical user interface, application

Description automatically generated

โดยไฟล์ที่ใช้จะชื่อว่า top100music.csv จะเก็บอยู่ใน program/assets

1. เมื่ออัพโหลดข้อมูลแล้ว

Table

Description automatically generated

**7**

**6**

**5**

**4**

**3**

**2**

**1**

(1) ช่องสำหรับค้นหาเพลงจะค้นหาตามชื่อ

(2) เมื่อค้นหาแล้วผลการค้นหาจะปรากฏที่ช่องนี้ให้ผู้ใช้เลือก โดยหลักการค้นหาจะค้นจากตารางด้านล่าง ดูจากชื่อของเพลงว่ามีคำที่เราค้นหาอยุ่หรือมไม่

Graphical user interface, application

Description automatically generated

(3) ปุ่มสำหรับกดเพื่อเลือกเพลงจากการค้นหาเมื่อกดแล้วจะแสดงแจ้งเตือนให้ยืนยัน ถ้ายืนยันโปรแกรมจะพาไปหน้าของเพลงนั้น (Music View)

Graphical user interface, text, application, chat or text message

Description automatically generated

(4) ตารางแสดงเพลงทั้งหมด ผู้ใช้สามารถกดเลือกเพลงได้เลยจะมีการทำงานเหมือน (3)

Graphical user interface, text, application, chat or text message

Description automatically generated

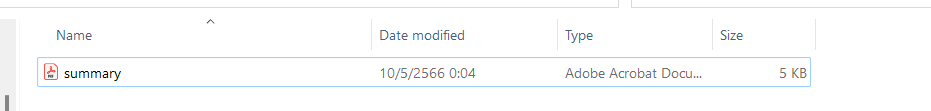
(5) Chart button โปรแกรมจะทำการนำผู้ใช้ไปที่ Chart View

(6) Report button เป็นฟังชันการพิมพ์สรุปข้อมูลโดยเมื่อกดจะมีแจ้งเตือนให้กดยืนยัน และเมื่อยืนยันจะมีแจ้งเตือนว่าสร้างไฟล์สำเร็จ จากนั้นโปรแกรมจะทำการเปิด Folder ตำแหน่งที่เก็บไฟล์รายงานมาให้

Graphical user interface, text, application, chat or text message

Description automatically generatedGraphical user interface, text, application, chat or text message

Description automatically generated



(7) Exit button

5. Chart View จะมีเมนูด้านซ้ายให้เลือกชนิดของ Chart และมีแถบการทำงานด้านบนสำหรับเลือกข้อมูล

Graphical user interface, application

Description automatically generated

ตัวอย่าง Scatter Chart เปรียบเทียบ year ต่อ popularity โดยแปรค่าได้ว่ามีเพลงที่ได้รับความนิยมสูงส่วนมากจะอยู่ในช่วงปี 2010-2020

Chart, scatter chart

Description automatically generated

ตัวอย่าง Artist-Genre Network Graph เป็นกราฟเครือข่ายแสดงให้เห็นว่าศิลปินนั้นมีความเชื่อมโยงกับชนิดของเพลงแบบไหน

A picture containing website

Description automatically generated

6. Music View จะเป็นหน้าที่แสดงข้อมูลรายละเอียดของเพลงนั้นโดยเมื่อทำการเลือกเพลงมา ถ้าเพลงนั้นยังไม่มีรูปในฐานข้อมูลก้จะแจ้งเตือนว่าไม่มีรูป

Graphical user interface, text, application, chat or text message

Description automatically generated

ในหน้านี้จะมีการแสดงข้อมูลของเพลงนั้นๆ และรูปภาพนอกเหนือจากนี้ยังมี ปุ่ม Open In youtube เพื่อเปิดเพลงนี้ใน youtube และปุ่ม Recommend Song เพื่อทำการแสดงเพลงที่แนะนำมา 5 เพลง

A picture containing graphical user interface

Description automatically generated

Recommend Song เมื่อทำการกดจะแจ้งเตือนให้ยืนยันในการแนะนำเพลง

Graphical user interface, text, application, chat or text message

Description automatically generatedGraphical user interface, text, application, chat or text message

Description automatically generated

7. การเพิ่มรูปภาพของแต่ละเพลงทำได้โดยการไปที่ program/assets/thumnail/ และทำการบันทึกภาพโดยชื่อภาพจะต้องตรงกับชื่อเพลงและเป็นนามสกุล gif

A picture containing graphical user interface

Description automatically generated

8. การเพิ่ม link ทำได้โดยการเพิ่มที่ไฟล์ top100music.csv ในคอลัมภ์ url

**อธิบายโค้ดแต่ละไฟล์ของโปรแกรม**

**โครงสร้างโปรแกรม**

**Graphical user interface, application

Description automatically generated**

**Main.py**

# Import necessary modules from tkinter

from tkinter import font

from tkinter import \*

from tkinter import ttk

# Import the UserController class from controller module

from controller import UserController

# Define the App class that initializes the main window and sets its properties

*class* App:

*def* \_\_init\_\_(*self*, *master*):

*self*.master = *master*

*self*.master.geometry("1360x770")  # Set the size of the window

*self*.master.resizable(False, False)  # Disable resizing the window

*self*.master.title("Recommend Music")  # Set the title of the window

        # Create a ttk style object to customize the appearance of the widgets

*self*.style = ttk.Style()

*self*.style.theme\_use('clam')  # Set the style to 'clam'

        # Create an instance of the UserController class to handle user interaction

*self*.controller = UserController(*self*.master)

        # Show a pop-up message to ask the user to upload a file to start the program

*self*.controller.popup("Caution", "Upload File in require to start program!")

        # Show the welcome view that displays the program's title and instructions

*self*.controller.show\_welcome\_view()

# If this module is executed as a script (not imported), create a new instance of the App class and start the main event loop

if \_\_name\_\_ == "\_\_main\_\_":

    root = Tk()

    app = App(root)

    root.mainloop()

ไฟล์นี้จะเป็นจุดเริ่มต้นของโปรแกรมมีการตั้งค่าเริ่มต้นของโปรแกรม เช่น ขนาดของหน้าต่าง และจะมีการสร้าง controller เพื่อเริ่มต้นโปรแกรม

**model.py**

# Define constants for file location, fonts, colors

FILE\_LOCATION = "./assets"

BUTTON\_FONT = ("Arial", 13, "bold")

LABEL\_FONT = ("Arial", 20, "bold")

LABEL\_FONT\_H1 = ("Arial", 36, "bold")

LABEL\_FONT\_H2 = ("Arial", 30, "bold")

USER\_FONT = ("Arial", 14, "bold")

INFO\_FONT = ("Arial", 12, "bold")

SMALL\_FONT = ("Arial", 12, "normal")

COLORS = ['green', 'red', 'purple', 'brown', 'blue']

MT\_FONT = ("Arial", 20, "bold")

AT\_FONT = ("Arial", 16, "bold")

# Import the pandas library to manipulate CSV files

import pandas as pd

# Define a Song class that represents a song object

*class* Song:

*def* \_\_init\_\_(*self*, *title*, *artist*, *top\_genre*, *year*, *bpm*, *energy*, *danceability*, *loudness*, *liveness*, *valence*, *length*, *acousticness*, *speechiness*, *popularity*, *url*):

*self*.title = *title*

*self*.artist = *artist*

*self*.top\_genre = *top\_genre*

*self*.year = *year*

*self*.bpm = *bpm*

*self*.energy = *energy*

*self*.danceability = *danceability*

*self*.loudness = *loudness*

*self*.liveness = *liveness*

*self*.valence = *valence*

*self*.length = *length*

*self*.acousticness = *acousticness*

*self*.speechiness = *speechiness*

*self*.popularity = *popularity*

*self*.url = *url*

*def* \_\_str\_\_(*self*):

        return *f*"{*self*.title} - {*self*.artist} ({*self*.year})"

*def* \_\_repr\_\_(*self*):

        return *self*.\_\_str\_\_()

ไฟล์น้าสำหรับกำหนด model ของเพลง และยังมีการกำหนด fonts สำหรับใช้งานในโปรแกรม

**Controller.py ไฟล์นี้เป็นแก่นของการทำงานโดยจะมีฟังชันต่างๆดังนี้**

**\_\_int\_\_()**

*self*.master = *master*

        # Data variables:

*self*.df = pd.DataFrame()

*self*.bar\_x\_label = StringVar()

*self*.bar\_y\_label = StringVar()

*self*.scatter\_x\_name = StringVar()

*self*.scatter\_y\_name = StringVar()

*self*.pie\_value\_name = StringVar()

*self*.pie\_group\_name = StringVar()

*self*.line\_name = StringVar()

*self*.currentMusic = ""

*self*.all\_songs = {}

*self*.default\_save\_path = os.path.join(os.getcwd(), "assets", "summary")

        # Bar box Label

*self*.bar\_box\_xlabels =[]

*self*.bar\_box\_ylabels =[]

        # Scatter box Label

*self*.scatter\_box\_xlabels =[]

*self*.scatter\_box\_ylabels =[]

        # Line box Label

*self*.line\_box\_labels =[]

        # Pie box Label

*self*.pie\_box\_xlabels =[]

*self*.pie\_box\_ylabels =[]

        # Setting Views

*self*.welcome\_view = WelcomeView(*self*.master, *self*)

*self*.chart\_view = ChartView(*self*.master, *self*)

*self*.music\_view = MusicView(*self*.master, *self*)

ทำหน้าที่ในการกำหนดแปรต่างๆที่สำคัญ ที่จำเป็นในโปรแกรม อีกทั้งยังมีการสร้าง Views ต่างๆเพื่อเรานำมาใช้งาน

**popup()**

*def* popup(*self*, *title*, *message*):

        messagebox.showinfo(*title*, *message*)

*def* decidePopup(*self*, *title*, *message*):

        confirmed = messagebox.askokcancel(*title*, *message*)

        return confirmed

1. The popup method takes two arguments - a title string and a message string - and displays a message box with the given title and message using the messagebox.showinfo method from the tkinter module. This method is used to display simple informational messages to the user.
2. The decidePopup method also takes a title string and a message string as arguments and displays a message box with the given title and message using the messagebox.askokcancel method from the tkinter module. This method prompts the user to make a decision by displaying "OK" and "Cancel" buttons. If the user clicks "OK", the method returns True. If the user clicks "Cancel", the method returns False. This method is used to prompt the user for confirmation before performing an action that could have consequences.

Show view functions

*def* show\_welcome\_view(*self*):

*self*.chart\_view.frame.pack\_forget()

*self*.music\_view.frame.pack\_forget()

*self*.welcome\_view.frame.pack()

*def* show\_chart\_view(*self*):

*self*.welcome\_view.frame.pack\_forget()

*self*.fill\_bar\_box()

*self*.fill\_line\_box()

*self*.fill\_pie\_box()

*self*.fill\_scatter\_box()

*self*.chart\_view.setComponent()

*self*.chart\_view.frame.pack()

*def* show\_music\_view(*self*):

*self*.welcome\_view.frame.pack\_forget()

*self*.chart\_view.frame.pack\_forget()

*self*.music\_view.show\_song\_view(*self*.currentMusic)

*self*.music\_view.frame.pack()

1. The show\_welcome\_view method hides the chart\_view and music\_view frames (if they are visible) and displays the welcome\_view frame by calling its pack method. This method is used to display the initial view of the program, which welcomes the user and prompts them to upload a file to begin.
2. The show\_chart\_view method hides the welcome\_view and music\_view frames (if they are visible) and displays the chart\_view frame by calling its pack method. Before displaying the chart view, this method calls several helper methods (fill\_bar\_box, fill\_line\_box, fill\_pie\_box, fill\_scatter\_box) to populate several UI elements on the chart view with data. It then calls the setComponent method of the chart\_view to update the UI with the new data.
3. The show\_music\_view method hides the welcome\_view and chart\_view frames (if they are visible) and displays the music\_view frame by calling its pack method. Before displaying the music view, this method calls the show\_song\_view method of the music\_view to populate the view with data for the current song (which is stored in the currentMusic attribute of the UserController). This method is used to display details about a particular song when the user selects it from the chart view.

**fileopen()**

*def* file\_open(*self*):

        file\_name = filedialog.askopenfilename(

*initialdir*=FILE\_LOCATION,

*title*="Open A File",

*filetypes*=(("csv files", "\*.csv"), ("All Files", "\*.\*"))

        )

        if file\_name:

            try:

                file\_name = *f*"{file\_name}"

*self*.df = pd.read\_csv(file\_name)

*self*.welcome\_view.enableBtn()

                # Create a dictionary of Song objects with the title as the key

*self*.all\_songs = {}

                for index, row in *self*.df.iterrows():

                    song = Song(

*title*=row["title"],

*artist*=row["artist"],

*top\_genre*=row["top genre"],

*year*=row["year"],

*bpm*=row["beats.per.minute"],

*energy*=row["energy"],

*danceability*=row["danceability"],

*loudness*=row["loudness.dB"],

*liveness*=row["liveness"],

*valence*=row["valance"],

*length*=row["length"],

*acousticness*=row["acousticness"],

*speechiness*=row["speechiness"],

*popularity*=row["popularity"],

*url*=row["url"]

                    )

*self*.all\_songs[song.title] = song

            except ValueError:

*self*.error\_info.config(*text*="file can not be opened!")

            except FileNotFoundError:

*self*.error\_info.config(*text*="file can not be found!")

*self*.clear\_table\_data()

        # from csv into dataframe:

*self*.my\_table["column"] = list(*self*.df.columns)

*self*.my\_table["show"] = "headings"

        for column in *self*.my\_table["column"]:

*self*.my\_table.heading(column, *text*=column)

        # resize columns:

        for column\_name in *self*.my\_table["column"]:

*self*.my\_table.column(column\_name, *width*=60)

        # fill rows with data:

        df\_rows\_old = *self*.df.to\_numpy()

        df\_rows\_refreshed = [list(item) for item in df\_rows\_old]

        for row in df\_rows\_refreshed:

*self*.my\_table.insert("", "end", *values*=row)

*self*.my\_table.place(*x*=5, *y*=5, *width*=1200, *height*=600)

1. The **filedialog.askopenfilename** method is called to prompt the user to select a CSV file from their system. The method takes the initial directory, title, and filetypes as parameters.
2. If a file is selected by the user, the CSV file is read into a pandas DataFrame and stored in the df attribute of the UserController. The file name is also stored in a local variable.
3. The **enableBtn** method of the **welcome\_view** is called to enable the button on the welcome view.
4. A dictionary of Song objects is created with the title of each song as the key, and the Song object as the value.
5. If the file cannot be opened or is not found, an error message is displayed in the UI.
6. The clear\_table\_data method is called to clear any existing data from the UI table.
7. The column names are set to the names of the columns in the DataFrame and are displayed in the UI table.
8. The width of each column in the table is set to 60 pixels.
9. Each row of the DataFrame is converted to a list and inserted into the UI table.
10. The UI table is placed at a specific location in the UI.

**Clear\_table\_data()**

* **For clear table data.**

**Fill chart function**

**Text

Description automatically generated**

In each of the methods (fill\_bar\_box, fill\_scatter\_box, fill\_pie\_box, and fill\_line\_box), the method first creates a list of column names using a list comprehension. Then, it iterates through each column and checks its data type. If the data type is 'object' (i.e., string), the column is added to the x\_labels list. If the data type is 'int64' or 'float64' (i.e., numeric), the column is added to the y\_labels list. Finally, the method sets the appropriate instance variables (bar\_box\_xlabels and bar\_box\_ylabels for the bar plot, scatter\_box\_xlabels and scatter\_box\_ylabels for the scatter plot, pie\_box\_xlabels and pie\_box\_ylabels for the pie plot, and line\_box\_labels for the line plot) to the x\_labels and y\_labels lists.

This method assumes that the DataFrame object has already been loaded into the instance variable "self.df" before calling any of these plot-specific methods.

**getrecommendMusicByName() – 1 (Use sklearn same vale but fast)**

*def* getRecommendMusicByName(*self*, *musicTitle*):

        # Define the features to use for similarity calculation

        features = ['beats.per.minute', 'energy', 'danceability', 'loudness.dB',

                     'liveness', 'valance', 'length', 'acousticness', 'speechiness']

        # Normalize the features

*self*.df[features] = *self*.df[features].apply(*lambda* *x*: (*x* - *x*.mean()) / *x*.std(), *axis*=0)

        # Compute the pairwise similarity matrix using cosine similarity

        sim\_matrix = cosine\_similarity(*self*.df[features])

        # Define a function to get recommendations based on a song

*def* get\_recommendations(*title*, *data*=*self*.df, *sim\_matrix*=sim\_matrix, *top\_n*=5):

            # Get the index of the song that matches the title

            idx = *data*[*data*['title'] == *title*].index[0]

            # Get the similarity scores of all songs with respect to the input song

            sim\_scores = list(enumerate(*sim\_matrix*[idx]))

            # Sort the songs based on the similarity scores

            sorted\_sim\_scores = sorted(sim\_scores, *key*=*lambda* *x*: *x*[1], *reverse*=True)

            # Get the top n most similar songs

            song\_indices = [i[0] for i in sorted\_sim\_scores[1:*top\_n*+1]]

            # Return the titles of the top n songs

            return *data*['title'].iloc[song\_indices].tolist()

        # Example usage

        return get\_recommendations(*musicTitle*)

"getRecommendMusicByName" that takes in a music title as input and returns a list of recommended music titles based on the similarity of their audio features to the input music title.

The method first defines a list of features to use for similarity calculation. These features include beats per minute, energy, danceability, loudness in decibels, liveness, valence, length, acousticness, and speechiness

Then, the method normalizes these features in the DataFrame object (self.df) by subtracting the mean and dividing by the standard deviation of each feature.

Next, the method computes the pairwise similarity matrix of the normalized features using cosine similarity. The similarity matrix will be a square matrix of size n x n, where n is the number of songs in the DataFrame object.

The method then defines a nested function named "get\_recommendations" that takes in the input music title, the DataFrame object, the similarity matrix, and the number of recommended songs to return (top\_n). The function first gets the index of the song that matches the input title in the DataFrame object.

It then computes the similarity scores of all songs with respect to the input song based on the cosine similarity values in the similarity matrix. The similarity scores are sorted in descending order, and the top\_n most similar songs (excluding the input song itself) are returned as a list of titles.

**getrecommendMusicByName() – 2 (hardcode)**

*def* getRecommendMusicByName(*self*, *musicTitle*):

        # Define the features to use for similarity calculation

        features = ['beats.per.minute', 'energy', 'danceability', 'loudness.dB', 'liveness', 'valance', 'length', 'acousticness', 'speechiness']

        # Normalize the features

*self*.df[features] = *self*.df[features].apply(*lambda* *x*: (*x* - *x*.mean()) / *x*.std(), *axis*=0)

        # Compute the pairwise similarity matrix using cosine similarity

        sim\_matrix = np.zeros((len(*self*.df), len(*self*.df)))

        for i in range(len(*self*.df)):

            for j in range(len(*self*.df)):

                if i != j:

                    vec1 = *self*.df.loc[i, features].values

                    vec2 = *self*.df.loc[j, features].values

                    sim\_matrix[i, j] = np.dot(vec1, vec2) / (np.linalg.norm(vec1) \* np.linalg.norm(vec2))

        # Define a function to compute the shortest paths between two nodes in a graph

*def* shortest\_paths(*matrix*):

            n = *matrix*.shape[0]

            for k in range(n):

                for i in range(n):

                    for j in range(n):

                        if *matrix*[i, k] + *matrix*[k, j] < *matrix*[i, j]:

*matrix*[i, j] = *matrix*[i, k] + *matrix*[k, j]

            return *matrix*

        # Compute the shortest paths between all pairs of songs

        dist\_matrix = shortest\_paths(1 - sim\_matrix)

        # Define a function to get recommendations based on a song

*def* get\_recommendations(*title*, *data*=*self*.df, *dist\_matrix*=dist\_matrix, *top\_n*=5):

            # Get the index of the song that matches the title

            idx = *data*[*data*['title'] == *title*].index[0]

            # Get the shortest distances to all other songs

            distances = *dist\_matrix*[idx, :]

            # Sort the songs based on the shortest distances

            sorted\_distances = sorted(enumerate(distances), *key*=*lambda* *x*: *x*[1])

            # Get the top n most similar songs

            song\_indices = [i[0] for i in sorted\_distances[1:*top\_n*+1]]

            # Return the titles of the top n songs

            return *data*['title'].iloc[song\_indices].tolist()

        # Example usage

        return get\_recommendations(*musicTitle*)

"getRecommendMusicByName" that takes in a music title as input and returns a list of recommended music titles based on the similarity of their audio features to the input music title.

The method first defines a list of features to use for similarity calculation, and then normalizes these features in the DataFrame object (self.df) by subtracting the mean and dividing by the standard deviation of each feature.

Instead of computing the pairwise similarity matrix using cosine similarity, this method calculates the pairwise distances between songs based on the normalized features. It initializes a matrix of zeros with the size of n x n, where n is the number of songs in the DataFrame object, and then loops through all possible pairs of songs except when i = j (i.e., the diagonal elements). For each pair, it calculates the dot product of their feature vectors and divides it by the product of their Euclidean norms to obtain the cosine similarity. It then subtracts the cosine similarity value from 1 to obtain the Euclidean distance between the songs, and stores the distance in the corresponding position of the distance matrix.

The method then defines a nested function named "get\_recommendations" that takes in the input music title, the DataFrame object, the distance matrix, and the number of recommended songs to return (top\_n). The function first gets the index of the song that matches the input title in the DataFrame object.

It then retrieves the shortest distances from the input song to all other songs in the distance matrix. The function sorts the distances in ascending order, and the top\_n least distant songs (excluding the input song itself) are returned as a list of titles.

Summarize\_df\_pdf() for create report file

*def* summarize\_df\_to\_pdf(*self*, *pdf\_path*=None):

        if not *self*.decidePopup("Caution", "Print Report to PDF!"):

            return

        summary = *self*.df.describe()

        if *pdf\_path* is None:

*pdf\_path* = os.path.join(*self*.default\_save\_path, 'summary.pdf')

        c = canvas.Canvas(*pdf\_path*, *pagesize*=A4)

        c.setFont('Helvetica-Bold', 18)

        c.drawString(50, 750, 'Summary of DataFrame')

        c.setFont('Helvetica', 12)

        x, y = 50, 700

        c.drawString(x, y, 'Attribute')

        c.drawString(x + 100, y, 'Count')

        c.drawString(x + 200, y, 'Mean')

        c.drawString(x + 300, y, 'Std')

        c.drawString(x + 400, y, 'Min')

        c.drawString(x + 500, y, '25%')

        c.drawString(x + 600, y, '50%')

        c.drawString(x + 700, y, '75%')

        c.drawString(x + 800, y, 'Max')

        y -= 20

        for col in summary.columns:

            x=50

            c.drawString(x, y, col)

            for val in summary[col]:

                c.drawString(x + 100, y, '{*:,.2f*}'.format(val))

                x+=100

            top\_songs = *self*.get\_top\_songs(col)

            if not top\_songs.empty:

                y -= 20

                c.drawString(50, y, 'Top 5 Songs:')

                y -= 20

                for \_, row in top\_songs.iterrows():

                    c.drawString(50, y, row['title'])

                    c.drawString(x + 100, y, '{*:,.2f*}'.format(row[col]))

                    y -= 20

            y -= 20

            # add a new page if the current page is filled with data

            if y <= 50:

                c.showPage()

                y = 700

        c.save()

*self*.popup("Success", "File has saved!")

        os.startfile(os.path.dirname(os.path.abspath(*pdf\_path*)))

"summarize\_df\_to\_pdf" that summarizes a pandas DataFrame object and saves the summary to a PDF file.

**If the "decidePopup"** method returns False, which indicates that the user has cancelled the operation, the method exits early. Otherwise, the method first computes the summary statistics of the DataFrame using the "describe" method and assigns the result to the "summary" variable.

**If no PDF file path** is provided, the method sets the path to a default location within the project directory. The method then initializes a new PDF document using the "canvas" module and sets the font styles for the document.

The method then loops through each column in the summary DataFrame and adds the column name and summary statistics to the PDF document. It also calls the "get\_top\_songs" method to get the top 5 songs for each column and adds them to the PDF document if they exist.

If the current page is filled with data, the method adds a new page to the document using the "showPage" method and resets the y-coordinate to the top of the page.

**WelcomeVIew.py**

เป็นฟังชันในการแสดงหน้าแรกของโปรแกรมมีการทำงานดังงี้

**\_init\_()**

****

"init" method initializes the instance variables, creates the main window, and places the widgets inside the window using the "place" geometry manager.

The top frame contains a label "Music Recommend" for the system name

The search bar contains a label "Name:", an entry field to input the search query, a list of past search queries, a combo box to display the list of past search queries, and a "Select" button to select a query from the list. The "search\_table" method is bound to the "<KeyRelease>" event of the entry field to update the table with the search results.

The left frame contains a table to display the music data. The table is implemented using the "ttk.Treeview" widget and is configured using the "configure" method of the "ttk.Style" class. The table is bound to the "<ButtonRelease-1>" event to retrieve the selected music title.

The button frame contains several buttons such as "Chart", "Report", "Get recommend music", and "Exit" to perform various actions in the system. The "Chart" button is bound to the "show\_chart\_view" method of the controller class to display the chart view. The "Report" button is bound to the "summarize\_df\_to\_pdf" method of the controller class to generate a summary report in PDF format. The "Get recommend music" button is not yet implemented and is currently disabled. The "Exit" button is bound to the "exitp" method to close the system.

**enableBtn()** method is used to enable various buttons and input fields of the GUI. When user importfile complete

*def* enableBtn(*self*):

*self*.buttonS.configure(*state*='normal')

*self*.button1.configure(*state*="normal")

*self*.button2.configure(*state*="normal")

*self*.button3.configure(*state*="normal")

*self*.search\_entry.configure(*state*="normal")

*self*.button0.place\_forget()

**print\_Title()**

*def* print\_title(*self*,*event*, *title*=None):

        if not *title*==None:

            pass

        else:

            item = *self*.controller.my\_table.selection()[0]

*title* = *self*.controller.my\_table.item(item, "values")[0]

*self*.controller. currentMusic = *title*

        if *self*.controller.decidePopup('Confirm', *f*'See Detail of {*title*}?'):

*self*.controller.show\_music\_view()

"print\_title" method is used to obtain the title of a music track from the Treeview widget in the GUI. It takes two parameters, "event" and "title", where "event" is the user action that triggers this method, and "title" is an optional parameter that can be passed to the method.

If "title" is not None, then the method does nothing. However, if "title" is None, the method extracts the selected item from the Treeview widget, obtains the value of the first column (i.e., the title of the music track), and assigns it to the "title" variable.

The "currentMusic" variable in the controller class is then set to the value of "title". If the user confirms their selection by clicking "OK" in the popup message, the "show\_music\_view" method of the controller class is called to display the detailed view of the selected music track.

Exitp()

For exit of program

*def* exitp(*self*):

        exit(0)

**Search\_table()**

*def* search\_table(*self*, *event*):

        items = *self*.controller.my\_table.get\_children()

        values = [*self*.controller.my\_table.item(item, 'values')[0] for item in items]

        search\_query = *self*.search\_entry.get().lower()

        filtered\_values = [value for value in values if search\_query in value.lower()]

*self*.combo['values'] = filtered\_values

        if filtered\_values:

*self*.combo.current(0)

"search\_table" method is used to implement a search functionality in the GUI. It is bound to the "<KeyRelease>" event of the search\_entry widget, which is triggered whenever a key is released while the search\_entry widget has the focus.

First, the method gets all the items in the Treeview widget using the "get\_children" method. Then, it extracts the value of the first column of each item using the "item" method and stores them in the "values" list.

Next, it gets the search query from the search\_entry widget and converts it to lowercase for case-insensitive search. It then filters the "values" list using a list comprehension to create a new list containing only the values that match the search query.

Finally, it updates the "values" of the combo box widget to the filtered values and sets the first filtered value as the current selection using the "current" method of the combo box widget. This updates the Treeview widget to show only the items that match the search query.

**ChartView.py** for show chart.

This code defines a class ChartView which creates a GUI for displaying various types of charts.

**\_\_init\_\_()**

\_\_init\_\_ method initializes the GUI components, including the top frame and buttons for selecting different chart types.

The setComponent method is called when a button is clicked and determines which chart to display based on the button clicked. It then calls a corresponding method to create and display the chart.

The buttonEx is a "Back" button that allows the user to return to the previous view.

**setComponent()**

Graphical user interface

Description automatically generated with medium confidence

setComponent method takes an argument ac which is used to determine which chart type is being set up. If ac is 0, the method does nothing. If ac is 1, it configures the chart components for a scatter chart; if ac is 2, it configures the chart components for a pie chart; if ac is 3, it configures the chart components for a line chart; if ac is 4, it configures the chart components for a network graph.

For each chart type, the method sets up the required chart components such as the x and y labels, the x and y comboboxes, and the draw and clear buttons. It also sets the values for the comboboxes based on the available chart data.

Finally, the method sets up the chart diagram by creating a canvas and placing it in the frame. The fig\_1 and output\_1 variables are used to store the chart figure and output respectively.

**Draw chart function**

Text

Description automatically generated

This code defines four functions that create different types of charts using the Python library matplotlib and a network chart using the networkx library. Each function takes input from a graphical user interface (GUI) built with tkinter and displays the chart in a tkinter canvas. Here is an overview of the four functions:

1. draw\_bar\_chart(): This function creates a bar chart using data from a pandas dataframe. The x-axis and y-axis labels for the chart are specified by the bar\_x\_label and bar\_y\_label variables in the GUI, respectively. The chart is created using the ax.bar() function and is styled using various matplotlib functions.
2. draw\_scatter\_chart(): This function creates a scatter plot using data from a pandas dataframe. The x-axis and y-axis variables for the chart are specified by the scatter\_x\_name and scatter\_y\_name variables in the GUI, respectively. The chart is created using the ax.scatter() function and is styled using various matplotlib functions.
3. draw\_pie\_chart(): This function creates a pie chart using data from a pandas dataframe. The data is grouped by a variable specified by the pie\_group\_name variable in the GUI. Categories that account for less than 2% of the data are combined into an "Other" category. The chart is created using the ax.pie() function and is styled using various matplotlib functions.
4. draw\_line\_chart(): This function creates a line chart using data from a pandas dataframe. The variable for the y-axis of the chart is specified by the line\_name variable in the GUI. The chart is created using the ax.plot() function and is styled using various matplotlib functions.
5. draw\_network\_chart(): This function creates a network chart using data from a pandas dataframe. The dataframe should have columns named 'artist' and 'top genre'. The chart is created using the networkx library and is styled using matplotlib functions.

**MusicView.py** for show single music

This code defines a class MusicView that represents a view in a graphical user interface (GUI) built with tkinter. The view displays information about a music song, including its attributes such as artist, top genre, and details, and provides buttons to open the song in YouTube and recommend other songs.

Here is an overview of the main parts of the code:

The \_\_init\_\_() function initializes the class and creates the GUI elements.

The style variable sets the style for the GUI using the ttk module.

The master variable is the main window of the GUI.

The controller variable is an instance of the controller class that manages the application's logic.

The frame variable creates a frame to hold the GUI elements.

The buttonEx variable creates a "Back" button that calls the show\_welcome\_view() method in the controller class when clicked.

The song\_label, artist, top\_genre, and detail variables create labels to display the song attributes and details.

The buttonLink variable creates a "Open in YouTube" button that calls the openWeb() method in the controller class when clicked.

The buttonR variable creates a "Recommend Songs" button that calls the recommend() method in the controller class when clicked.

Overall, this class provides a view for the user to interact with the application and access the song information and features.

**Show\_song()**

*def* show\_song\_view(*self*, *song\_title*):

        # Get the Song object with the given title from the all\_songs dictionary

        song = *self*.controller.all\_songs.get(*song\_title*)

        if song:

            # Set the text of the song\_label to display all the attributes of the Song object

*self*.song\_label.config(*text*=*f*"Title: {song.title} [{song.year}]")

*self*.artist.config(*text*=*f*"Artist: {song.artist}")

*self*.top\_genre.config(*text*=*f*"Genre: {song.top\_genre}")

*self*.detail.config(*text*=*f*"Detail\n"

*f*"BPM: {song.bpm}\n"

*f*"Energy: {song.energy}\n"

*f*"Danceability: {song.danceability}\n"

*f*"Loudness: {song.loudness}\n"

*f*"Liveness: {song.liveness}\n"

*f*"Valence: {song.valence}\n"

*f*"Length: {song.length}\n"

*f*"Acousticness: {song.acousticness}\n"

*f*"Speechiness: {song.speechiness}\n"

*f*"Popularity: {song.popularity}\n"

*f*"URL: {song.url}")

            try:

*self*.image = PhotoImage(*file*=*f*"assets/thumnail/{song.title}.gif")

*self*.imageShow  = Label(*self*.frame, *image*=*self*.image, *width*=800, *height*=700)

*self*.imageShow .place(*x*=500, *y*=80)

            except:

*self*.controller.popup("Warning", "This song don't have Image")

*self*.image = PhotoImage(*file*=*f*"assets/thumnail/none.gif")

*self*.imageShow  = Label(*self*.frame, *image*=*self*.image, *width*=800, *height*=700)

*self*.imageShow .place(*x*=500, *y*=80)

        else:

*self*.song\_label.config(*text*="Song not found!")

show\_song\_view that takes in a song\_title parameter.

The method first tries to find the Song object with the given title from a dictionary called all\_songs that is stored in the controller object.

**If the Song object** is found, the method updates various tkinter Label widgets with information about the song, such as the song's title, artist, genre, and details such as tempo, energy, danceability, and so on.

The method also tries to load a thumbnail image of the song from a file named {song\_title}.gif in the assets/thumnail directory. If the image file exists, it is displayed in a tkinter Label widget. Otherwise, a warning message is displayed and a default "none.gif" image is loaded and displayed.

**If the Song object** is not found, the method updates a tkinter Label widget to display a "Song not found!" message.

**openWeb()**

*def* openWeb(*self*):

*self*.sheet = Sheet(*self*.frame, *width*=800, *height*=400)

*self*.sheet.enable\_bindings()

        try:

*self*.sheet.set\_sheet\_data([[webbrowser.open(*self*.controller.all\_songs.get(*self*.controller.currentMusic).url)]])

        except:

*self*.controller.popup("Warning", "This song don't URL")

The method creates an instance of a Sheet object, which is a custom tkinter widget used to display tabular data. The widget is placed within the frame of the GUI, and its bindings are enabled.

The method then tries to retrieve the URL of the current music from the all\_songs dictionary stored in the controller object. If the URL exists, the webbrowser.open() function is used to open the URL in a web browser window.

The URL is wrapped in a list and set as the data for the Sheet object. This means that a single cell will be displayed in the Sheet widget, and clicking on that cell will open the URL in a web browser window.

If the URL is not found for the current music, the method displays a warning message using the popup() method of the controller object.

**Recommend()**

*def* reccommend(*self*):

        if *self*.controller.decidePopup('Confirm', *f*'Get reccomend music?'):

*self*.controller. show\_loading\_popup()

*self*.master.update()

            recommended\_songs = *self*.controller.getRecommendMusicByName(*self*.controller.currentMusic)

*self*.controller.hide\_loading\_popup()

*self*.controller.popup("Top 5 Recommended Songs", "\n".join(recommended\_songs))

This code defines a method called recommend that is used to get a list of recommended songs based on the current music being displayed in the GUI.

The method first checks if the user wants to get recommended music by using the decidePopup() method of the controller object to display a confirmation message. If the user confirms, the method proceeds to get the recommended music.

Before retrieving the recommended music, the method displays a loading popup using the show\_loading\_popup() method of the controller object. This is done to inform the user that the process of getting recommended music is in progress.

The method then calls the getRecommendMusicByName() method of the controller object to get a list of recommended songs based on the current music.

Once the recommended songs have been retrieved, the loading popup is hidden using the hide\_loading\_popup() method of the controller object.

Finally, the method displays a popup message using the popup() method of the controller object to show the top 5 recommended songs. The recommended songs are joined together with newline characters (\n) and displayed in the popup message.