Final Report for Project 3

https://github.com/joemackle/MixtapeEngine

Group 150:

Eric Fontes	Joseph Mackle	Neil Patel
https://github.com/kindergartener	https://github.com/joemackle	https://github.com/Neil-Patel-12

Video Demo:

https://youtu.be/SZUcBBey6pc

Proposal

Problem: The Spotify application only has the ability to search for artists, a particular song title, and music genre. With the mixes feature, users can search for an emotion and get an algorithmically generated playlist for them such as a “Happy Mix” or a “Joyful Mix”. However, if a user wanted to generate a playlist that combines multiple emotions, they would be out of luck–Spotify does not support the creation of a “Happy Sad Mix” or a “50% Dance 50% Acoustic Mix”.

Motivation: As a consequence of the problem above, we created the Mixtape Engine, a web app that allows users to generate playlists based on a set of user-defined emotional values. The motivation for creating this project is inspired by a common thread shared among all music fans: the desire to find new music. Our solution aims to provide a unique and consistent way of introducing users to new music based on their emotional state.

Features: The main features of the web app are located in the cards visible in the home page. The first card contains sliders that correspond with each of the seven emotional qualifiers (danceability, energy, etc.). The user inputs their desired parameters and chooses a sorting algorithm from the drop-down menu, then presses the “Parse CSV” button which sorts the songs in the database using the selected algorithm. The elapsed time is displayed below the button. In the second card, the songs generated by the algorithm are displayed in a playlist format. On the right, the user can specify if they want the playlist sorted by artist, best fit, or song name. Clicking on a song in the playlist will open a Spotify link to that song.

Data: The data for this project was obtained from a Hugging Face dataset published by user maharshipandya: https://huggingface.co/datasets/maharshipandya/spotify-tracks-dataset. Collected using Spotify’s Web API and Python, the dataset is a csv file with tracks spanning 125 genres. Each row of the

csv file corresponds to a track, with extra dimensions holding information about the song name, artist, genre, tempo, and more. Of particular interest to our group were the columns: Danceability, Energy, Speechiness, Acousticness, instrumentalness, Liveness, and Valence. These columns are values from 0 to 1 and represent qualitative measures associated with each song. For example, a Bach prelude would score close to 0 on the energy scale whereas a death metal song would score close to 1. The PapaParse javascript library was used to parse this csv file and convert it into a javascript array for ease of access.

Technologies used:

Languages	Frameworks	Libraries / APIs
Javascript, HTML, CSS	SvelteKit, TailwindCSS, Flowbite	PapaParse

Algorithms implemented: For the Mixtape Engine, we implemented 6 different sorting algorithms. The algorithms we used are selection sort, bubble sort, insertion sort, shell sort, merge sort, and quick sort. These sorting algorithms take in, as input, a data array from the parseCsv.js file and sorts that array in place in descending order (greatest to least) with respect to the score property. After the chosen algorithm is done sorting the array, based on the "score", the time, in seconds, gets displayed and the top 10 songs get presented to the user based on the 7 emotions the user imputed. The songs are displayed in a vertical fashion with the name/ title of the song, is on top and the artist/musician/band right below said title.

Additional data structures/algorithms: Additionally, we created our own algorithm which is located in the parseCsv.js file. This additional algorithm creates a new property entry in every row called "score". The score is calculated by iterating through every song in the array and generating a new value based on a linear combination of the seven qualitative measures (danceability, energy, etc.) and the corresponding weights obtained through user input. The formula is described below:

$$\text{score} = \sum_{i=1}^{7} Q_i W_i \qquad Q = \{\text{song.danceability, song.energy, } \cdots\}$$
$$W = \{\text{weights.danceability, weights.energy, } \cdots\}$$

If the user pushes the danceability weight slider all the way up to 1 while keeping all others at 0, the playlist will sort by danceability in descending order. If the user pushes the danceability and energy sliders to 1 while keeping all others at 0, the playlist will be sorted based on a mixture of danceability and energy. This is the procedure we use to sort songs based on arbitrary emotional values.

Distribution of roles: Responsibilities were divided evenly to ensure every member had an equal workload. Joseph was in charge of managing the Github repository as well as creating a frontend and designing graphics for the website. Eric was in charge of parsing and manipulating the database as well as the management of the score metric for sorting. Finally, Neil was in charge of implementing all the sorting algorithms in javascript as well as connecting these algorithms with the csv file parse function.

Analysis

Changes made after proposal: There were no major changes made after the proposal. We knew this was going to be a big project that would require a lot of work on both the back-end and front-end. We also know that we wanted to use JavaScript/HTML/CSS to make a web application.

Time complexity:

- Selection Sore: The worst, average, and best case time complexity is O(n^2), where n is the number of songs in the array.
    - Key takeaway: starting from the first index to the last index, we select the greatest "score" and swap it with the first array entry. We select and swap for each entry.
- Bubble Sort: The worst and average case time complexity is O(n^2), where n is the number of songs in the array. The best case time complexity is O(n) where n is the number of songs in the array. This will happen if the array is closely sorted in descending order.
    - Key takeaway: This algorithm compares the 2 adjacent elements and waps them in place if the second element is greater than the other. When no swaps occur, that is when the algorithm is done.
- Insertion Sort: The worst and average case time complexity is O(n^2), where n is the number of songs in the array. The best case time complexity is O(n), where n is the number of songs in the array. This will only happen if the array is closely sorted in descending order.
    - Key takeaway: The beginning of the array is sorted and for each element that is not sorted, we insert it into the array in its sorted position.
- Shell Sort: The worst case time complexity is O(n^2), where n is the number of elements/ songs in the array (worst case will occur if a terrible gap increments are used). The best case time complexity will result if the best possible gap is chosen being O(n*log(n)), where n is the number of elements/ songs in the array. This is because with the best gap sequence, the array is broken into smaller subarrays that are almost sorted, making the insertion sort step very efficient. The average case time complexity of Shell sort is still a matter of theoretical analysis, as it heavily depends on the choice of gap sequence. Some gap sequences have been found to have an average case time complexity of O(n^(3/2)), but there's no definitive average case time complexity for all possible sequences.
    - Key takeaway: This algorithm improves upon the insertion sort algorithm. It works by sorting the elements at a specific interval, gradually reducing the interval until the entire array is sorted. The interval is commonly called the "gap".
- Merge Sort: The worst, average, and best case time complexity for this algorithm is O(n*log(n)), where n is the number of elements/ songs in the array. Merge sort's consistent O(n log n) time complexity across all cases makes it a popular choice for sorting large datasets, especially when stability and predictable performance are important factors. However, it does require additional memory space proportional to the size of the input array for the temporary arrays used during the merge step.
    - Key takeaway: Merge sort is a divide-and-conquer algorithm that divides the input array into smaller subarrays, sorts them independently, and then merges them back together into a single sorted array.

- Quick Sort: The worst case time complexity is O(n^2), where n is the number of elements/ songs in the array. This occurs when the pivot selection strategy consistently picks either the smallest or largest element in the array. In such a scenario, one partition would contain all elements except the pivot, leading to highly unbalanced partitions. This can result in a recursion tree with height close to 'n'. The best and average case time complexity of Quick sort occurs when the pivot element is always chosen such that it divides the array into two equal halves. In this scenario, each partition step divides the array into two nearly equal-sized subarrays. As a result, the recursion tree becomes balanced, and the time complexity is O(n log n), where 'n' is the number of elements in the array. This scenario is less likely to happen randomly, but it's common when the pivot is chosen as the median element.
    - Key takeaway: Quick sort often outperforms other sorting algorithms like Merge sort due to its cache efficiency and lower constant factors. It's widely used in libraries and programming languages for sorting large datasets efficiently. Highly favored for its average and best-case performance, which are O(n*log(n)).

Reflection

As a group, the overall experience for the project can best be described as a learning experience. While we had worked with Git version control before, we cemented the process of using Git in our workflows through this project. Furthermore, this was a great introductory experience to learn Svelte as a means of web development.

One challenge we faced during this project was the interaction between client-side processing and sorting a dataset of 114,000 data points. When parsing the entire dataset, the quadratic sorts took too much time and crashed the web browser. As a result, we restricted our input size to 25,000 data points in order for the browser to handle the request properly. Although we processed and created all 114,000 data points, we only sorted a subset of 25,000 of them.

References

- Dataset: https://huggingface.co/datasets/maharshipandya/spotify-tracks-dataset
- Papaparse documentation: https://www.papaparse.com/
- SvelteKit documentation: https://kit.svelte.dev/docs/introduction
- Flowbite component library: https://flowbite.com/#components