Spotify Konsolenanwendung – Dokumentation

Ein Projekt von:

Ben van Lier

Betreuer:

Jürgen Toth

Abgabedatum:

27.09.2024

Data Science und künstliche Intelligenz

WDS23A

Duale Hochschule Baden-Württemberg (DHBW)
Campus Lörrach

1. Projektbeschreibung und Vorgehen

1.1 Anforderungen, Systemarchitektur und Datenstruktur

Zu Beginn des Projekts wurde ein Programm entwickelt, das auf Benutzereingaben reagiert und eine ähnliche Funktionalität wie Spotify bietet. Das Ziel war es, ein System zu schaffen, das verschiedene Informationen zu Songs erfasst, darunter den Titel, das Album, den Interpreten sowie das Jahr der Veröffentlichung. Die Hauptanforderung bestand darin, dass das Programm es ermöglicht, neue Lieder in eine Liste, die sogenannte "Library", aufzunehmen und bestehende Songs bei Bedarf wieder zu entfernen, aber vor allem sollten Funktionen zur Suche und Sortierung implementiert werden, um den Zugriff auf bestimmte Lieder effizient zu gestalten. Ein weiterer wichtiger Aspekt war die Möglichkeit, eine Liste von Favoriten zu pflegen, indem Songs als Favoriten markiert und wieder entfernt werden können.

Ein weiterer Punkt war die Überlegung, eigene Playlists erstellen zu können, anstatt nur mit der allgemeinen Library zu arbeiten. Dies wurde als nützlich erachtet, um eine realistischere Benutzererfahrung zu bieten, ähnlich dem tatsächlichen Umgang mit Musik-Apps. Mit dieser Funktion können Benutzer Playlists nach ihren Vorlieben zusammenstellen und Songs zwischen diesen hin- und herbewegen. Das Abspielen von Songs in das Programm zu integrieren ist nicht Teil des Projekts, da der Fokus auf der Verwaltung und Organisation der Song-Daten liegt.

Architektur und Funktionalitäten

Das Projekt wurde in Python geschrieben und ist in drei zentrale Klassen unterteilt: die Klasse Song, die Klasse Playlist und die Klasse Library. Die Song-Klasse ist dafür zuständig, aus den eingegebenen Song-Daten Objekte zu erzeugen. Jedes dieser Objekte enthält die relevanten Informationen wie Titel, Interpret, Album und Jahr. Diese Objekte werden dann in der Library gespeichert, die als zentrales Verwaltungselement dient. [1] In der Library erfolgen alle wesentlichen Funktionen wie das Suchen, Sortieren und Verwalten der Song-Objekte. Zusätzlich enthält die Library auch Mechanismen zur Verwaltung von Favoriten und Playlists.

Die Playlist-Klasse ist vergleichsweise einfach aufgebaut. Sie bietet grundlegende Funktionen zum Hinzufügen (add_song) und Entfernen (remove_song) von Songs in den erstellten Playlists. Alle Funktionen, die das Design und die Navigation zwischen den Menüs betreffen, sowie die Benutzerinteraktionen zur Auswahl der nächsten Aktion, werden außerhalb der Klassen definiert. Diese Menüstrukturen wurden im Laufe des Projekts verfeinert, um eine kompaktere und benutzerfreundlichere Menüführung zu gewährleisten.

[1] Library: Die Hauptliste aller am Start des Programms instanziierter Songs, auf die jegliche Such und Sortieralgorithmen zugreifen

Datenquelle und Datenaufbereitung

Die Song-Daten werden aus einer CSV-Datei geladen (songs.csv), sobald das Programm ausgeführt wird. Diese Datei dient als statische Datenquelle, aus der die Songs in die Library importiert werden. Anzumerken ist, dass die CSV-Datei durch den Benutzerinput im laufenden Programm nicht verändert wird. Änderungen wie das Hinzufügen oder Entfernen von Songs erfolgen nur temporär in der aktuellen Laufzeit des Programms innerhalb der Library. Bei einem Neustart des Programms wird die Library erneut aus der unveränderten CSV-Datei geladen, was bedeutet, dass alle während der Nutzung vorgenommenen Änderungen nicht dauerhaft gespeichert werden.

Die verwendeten Daten sind ursprünglich aus einer CSV-Datei, [2] die für die Verarbeitung im Hauptprogramm spotify.py aufbereitet wurden. Dabei wurden durch data_polisher.py überflüssige Zeichen und Spalten der originalen CSV entfernt, die bei der ausschließlichen Verarbeitung von title, artist, album und year unvorteilhaft wären. Alle Jahreszahlen die als Null angegeben wurden, setzt das data_polisher.py auf einer Zufallszahl zwischen 1930 und 2010 um im realistischen Rahmen der Songdaten mit valider Jahreszahl zu bleiben.

1.2 Implementierungsschritte und Algorithmen [3]

Nachdem ein Skelett mit allen Kernfunktionen vorhanden war, wurden diese stückweise mit tatsächlicher Funktionalität gefüllt. Zuerst simple Funktionen, wie zum Beispiel add_song_to_library und delete_song_from_library, welche, wie ihre Namen suggerieren, das Hinzufügen und Entfernen von einzelnen Songs zur oder von der Hauptliste ermöglichen. Danach folgte die Implementierung von weiteren essentiellen Funktionen zur Navigation durch Benutzerinput. Im Anschluss wurden die Funktionen zum Verwalten von Favoriten und dem Erstellen und Verwalten von benutzerkreierten Playlists programmiert, was das Ende der simplen Funktionsimplementierung beschreibt.

Komplexere Implementierung: Such- und Sortieralgorithmen

Nach den Grundfunktionen konzentrierte sich die Entwicklung auf die komplexeren Teile des Programms: die Such- und Sortieralgorithmen. Zunächst wurden die grundlegenden Algorithmen für lineare und binäre Suchen sowie Insertion-, Selection- und Quick-Sort implementiert. [4] Diese Algorithmen mussten so gestaltet werden, dass sie auf alle relevanten Felder der Songdaten angewendet werden können: title, artist, album, year.

Eine besondere Herausforderung war dabei die korrekte Implementierung der Such- und Sortierlogik (search_songs und sort_library), um verschiedene Datentypen (Strings für Titel, Künstler und Alben sowie Integer für Jahreszahlen) zu unterstützen.

---

[2] Noch nicht aufbereitete Datenquelle: https://github.com/mahkaila/songnames/blob/master/SongCSV.csv

[3] Die Implementierung aller Funktionen enthält verständlicherweise Schrittweise Tests, um die Funktionalität des gesamten Programms nach einzelnen Änderungen zu Garantieren

[4] Code der Algorithmen: https://youtu.be/Nkw6Jg_Gi4w, https://youtu.be/mI3KgJy_d7Y, https://youtu.be/CB_NCoxzQnk, https://youtu.be/zeULw-a7Mw8, https://youtu.be/wNOoyZ45SmQ

Hierbei musste auch bedacht werden, dass der Vergleich von Werten je nach Attribut unterschiedlich behandelt wird (z.B. String-Vergleich vs. numerischer Vergleich). Die Algorithmen an sich sind in den Funktionen insertion_sort, selection_sort, quicksort, binary_search und linear_search implementiert. Handhabung von verschiedenem Datentypen dagegen in search_songs und sort_library.

Ein wesentlicher zeitaufwändiger Teil der Implementierung war die Sicherstellung, dass das Programm robust gegenüber fehlerhaftem Benutzerinput ist. Bei der Interaktion mit Benutzern können verschiedene Arten von Fehleingaben auftreten, wie beispielsweise falsche Datentypen (z.B. Eingabe eines Strings anstelle einer Zahl). Dies wurde durch den Einsatz von try-except-Blöcken und while-Schleifen erreicht, die es ermöglichen, Fehler abzufangen und den Benutzer um eine erneute Eingabe zu bitten.

2. Auswahl der Algorithmen und Laufzeitanalyse

Ein zentrales Element der Spotify-Konsolenanwendung ist die Implementierung von Such- und Sortieralgorithmen, die es dem Benutzer ermöglichen, effizient auf die gespeicherten Songs zuzugreifen und diese zu verwalten. Bei der Auswahl der Algorithmen wurde auf die spezifischen Anforderungen der Anwendung, insbesondere der Datentypen und des Benutzerinputs, geachtet. In diesem Kapitel werden die verwendeten Sortier- und Suchalgorithmen erläutert. Ihre theoretischen Laufzeiteffizienzen werden mittels O-Notation dargestellt, während die tatsächlichen Laufzeiten anhand eines finalen Datensatzes von 10.000 Songs manuell gemessen wurden. [5] [6]

2.1 Laufzeitanalyse der Sortieralgorithmen

Um die Songs in der Library nach verschiedenen Kriterien wie title, artist, album oder year zu sortieren, wurden drei Sortieralgorithmen implementiert: Insertion-Sort, Selection-Sort und Quick-Sort. Diese Algorithmen unterscheiden sich sowohl in ihrer theoretischen Laufzeiteffizienz als auch in ihrer tatsächlichen Performance, gemessen am finalen Datensatz.

Insertion-Sort ist ein einfacher Algorithmus, der jedes Element der Liste mit den vorherigen vergleicht und es an der richtigen Position einfügt. Die theoretische Laufzeitkomplexität von Insertion-Sort liegt im schlechtesten Fall bei $O(n^2)$. Obwohl Insertion-Sort für kleinere, fast sortierte Listen eine gute Wahl wäre, wurde er hier auf dem finalen Datensatz von 10.000 Songs getestet, um einen Vergleich zu geeigneteren Algorithmen zu ermöglichen. Die tatsächliche Laufzeit betrug für title, artist oder album im Durchschnitt 18.87 Sekunden und für year 10.49 Sekunden, was die $O(n^2)$-Laufzeit auf diesem unsortierten, verhältnismäßig großen Datensatz bestätigt. Der Selection-Sort-Algorithmus hat ebenfalls eine theoretische Laufzeit von $O(n^2)$, da er in jeder Iteration das kleinste Element auswählt und an die richtige Position verschiebt, was den Algorithmus ebenfalls weniger effizient für große Datensätze macht. Die gemessene Laufzeit betrug im Durchschnitt 36.33 Sekunden für title, artist oder album und 14.59 Sekunden für year, was die ineffiziente Natur dieses Algorithmus bei größeren Datensätzen verdeutlicht. Um die Sortierung des vorliegenden großen Datensatzes in akzeptabler Zeit durchzuführen, wurde Quick-Sort eingeführt, ein Algorithmus mit einer durchschnittlichen Laufzeit von $O(n \log n)$. Er basiert auf dem "Divide and Conquer"-Prinzip und teilt die Liste rekursiv in kleinere Teilmengen. Beim Test auf dem finalen Datensatz wurde eine Laufzeit von 0.31 Sekunden für title, artist, album und 0.06 Sekunden für year gemessen, was die theoretische Effizienz dieses Algorithmus unter realen Bedingungen bestätigt.

[5] 10000 Songs, nicht aufbereitet: https://github.com/mahkaila/songnames/blob/master/SongCSV.csv

[6] Die Laufzeit wird bei jedem abgeschlossenen Such- und Sortierprozess ausgegeben, was das Testen erleichtert und Benutzer Feedback gibt. Die Messung ist durch Verwendung des time Moduls möglich: https://docs.python.org/3/library/time.html

2.2 Laufzeitanalyse der Suchalgorithmen

Neben den Sortieralgorithmen spielen auch die Suchalgorithmen eine entscheidende Rolle in der Effizienz der Spotify-Konsolenanwendung. Es wurden zwei Suchalgorithmen implementiert: Lineare Suche und Binäre Suche. Beide Algorithmen unterscheiden sich in ihrer theoretischen Laufzeit und zeigen in der Praxis bei verschiedenen Attributen unterschiedliche Laufzeiten.

Die lineare Suche durchläuft jedes Element der Liste nacheinander und vergleicht es mit dem gesuchten Attribut. Dieser Algorithmus hat eine theoretische Laufzeit von O(n), was bedeutet, dass die Suchzeit linear mit der Größe der Liste ansteigt. In den durchgeführten Tests wurde die lineare Suche auf einem finalen Datensatz von 10.000 Songs für verschiedene Attribute wie title, artist, album und year durchgeführt.

Die durchschnittliche Laufzeit für zufällige Suchanfragen nach title, artist oder album betrug 0,002 Sekunden, während die Suche nach year schneller war und im Durchschnitt 0,001 Sekunden dauerte. Der Grund für diese Unterschiede liegt hauptsächlich darin, dass der Vergleich von numerischen Werten wie year schneller abläuft als der Vergleich von Strings bei den Attributen title, artist und album. [7]

Die binäre Suche hingegen ist theoretisch effizienter als die lineare Suche, da sie die Liste voraussetzt, dass sie bereits sortiert ist, und dann schrittweise die Menge der zu durchsuchenden Elemente halbiert. Dies führt zu einer theoretischen Laufzeit von O(log n), was bei großen Datensätzen eine erheblich schnellere Suche ermöglicht. In den Tests wurde die binäre Suche zusammen mit einer Sortierung durch Quick-Sort der Liste auf verschiedenen Attributen wie title, artist, album und year durchgeführt. Das gestaltet die insgesamte Laufzeit natürlich langsamer, da die Laufzeit der Sortierung mindestens in der ersten Suche nach dem gewünschten Attribut miteinberechnet wird. Auffällig wird im Folgenden, dass die tatsächliche Laufzeit der binären Suche höher ist, als die der linearen Suche, was an dem im Kontext von Suchalgorithmen vergleichsweise kleinen Datensatz zu erklären ist. Die binäre Suche wird in der Regel erst bei Datensätzen in der Größenordnung von mehreren Millionen Einträgen effizienter.

Die durchschnittliche Laufzeit der binären Suche betrug 0,003 Sekunden für zufällige Suchanfragen nach title, album und year, wobei insbesondere bei überrepräsentierten Jahreszahlen wie 1999 und 2000 eine leicht abweichende Laufzeit beobachtet wurde. Ein bemerkenswerter Effekt, der bei den Tests der binären Suche auftrat ist, dass nach der ersten Sortierung die nachfolgenden Suchanfragen in derselben Kategorie (z.B. mehrere Suchanfragen nach year) schneller durchgeführt wurden, da die Liste bereits sortiert war.

Die binäre Suche etwas funktioniert schneller bei den häufig vorkommenden Werten wie 1999 und 2000, weil diese tendenziell näher an der Mitte des sortierten Datensatzes liegen oder häufiger gefunden werden, was die Anzahl der notwendigen Vergleichsschritte verringert.

[7] https://stackoverflow.com/questions/4904179/why-is-integer-comparison-faster-then-string-comparison

2.3 Einflussfaktoren auf die Laufzeit [8]

Die tatsächliche Performance von Such- und Sortieralgorithmen wird wie zu erkennen ist, also nicht nur von der O-Notation bestimmt, sondern von einer Vielzahl praktischer Faktoren beeinflusst, die von der Art der Daten über ihre Verteilung bis hin zur zugrunde liegenden Hardware reichen. Für die Spotify-Konsolenanwendung erwiesen sich Quick-Sort und die lineare Suche in den meisten Fällen als effizienteste Algorithmen, während Insertion- und Selection-Sort vor allem bei unsortierten und komplexeren Datentypen wie Strings langsamer waren. Die binäre Suche wäre definitiv effizienter, falls der Datensatz deutlich vergrößert wird. Diese Ergebnisse verdeutlichen, wie wichtig es ist, bei der Auswahl von Algorithmen nicht nur die theoretische Komplexität zu berücksichtigen, sondern auch die realen Bedingungen und Anforderungen der Anwendung zu verstehen.

[8] Alle Laufzeittests, Anmerkungen zu verschiedenen Szenarien und dessen Resultate sind im Dokument tests.txt einsehbar

3. Änderungen für Deployment:

3.1 Optimierung der Datenquelle und Algorithmen

Um die Spotify-Konsolenanwendung auf eine reale Benutzerumgebung zu erweitern, sind mehrere wichtige Änderungen erforderlich, um sowohl die Datenquelle zu vergrößern als auch die Benutzererfahrung zu optimieren. Der erste Schritt wäre, die aktuelle CSV-Datei mit 10.000 Songs gegen eine größere Datenquelle auszutauschen, die dem Benutzer eine größere Auswahl bietet. Auch wenn CSV weiterhin als Format verwendet werden kann, sollte die Datenmenge so erhöht werden, dass sie der Realität einer umfangreichen Musiksammlung näherkommt. Mit dieser erweiterten Datenmenge würde sichergestellt, dass die binäre Suche gegenüber der linearen Suche einen deutlichen Leistungsvorteil bietet, da sie speziell bei großen Datensätzen effizienter ist. Dies rechtfertigt die Implementierung der binären Suche als einzige Suchmethode.

In Kombination mit der binären Suche sollte Quick-Sort der einzige Sortieralgorithmus sein, da er eine deutlich bessere Performance bei großen Datensätzen aufweist (sogar schon bei der aktuell implementierten Datenquelle) als die implementierten Algorithmen Insertion- und Selection-Sort, die sich bei kleineren, fast sortierten Daten bewähren, aber für reale Musiksammlungen ineffizient sind.

Ein weiterer wichtiger Aspekt für das Deployment wäre, die Datenquelle dynamisch zu gestalten. Derzeit bleibt die CSV-Datei unverändert, wenn der Benutzer Songs hinzufügt oder entfernt, nur die Library wird während der Laufzeit des Programms bearbeitet. Für eine realistische Anwendung wäre es jedoch notwendig, dass jede Änderung an der Datenbank direkt in der Quelle gespeichert wird, sodass sie auch nach einem Neustart der Anwendung verfügbar bleibt. Dies könnte durch ein dynamisches Aktualisieren der CSV-Datei oder durch die Verwendung einer Datenbank wie SQLite erreicht werden.

Darüber hinaus sollten auch Playlists und Favoriten permanent gespeichert werden. In der aktuellen Version werden diese Änderungen beim Schließen der Anwendung zurückgesetzt, was für einen echten Benutzer nicht akzeptabel wäre. Die Anwendung müsste so erweitert werden, dass Playlists und Favoriten dauerhaft gespeichert und beim Neustart der Anwendung automatisch geladen werden.

Zusammengefasst würden diese Änderungen die Anwendung auf ein Deployment vorbereiten, das sowohl in Bezug auf die Performance der Algorithmen als auch auf die Benutzerfreundlichkeit optimiert ist. Eine größere, dynamische Datenquelle, die dauerhafte Speicherung von Änderungen und die Fokussierung auf effiziente Algorithmen wie Quick-Sort und binäre Suche würden eine benutzerfreundliche und effiziente Anwendung sicherstellen.

3.2 Grund für den Verzicht auf die vorgeschlagenen Änderungen

Die in diesem Kapitel vorgeschlagenen Änderungen wurden nicht in die aktuelle Implementierung aufgenommen, da das primäre Ziel dieses Projekts darin besteht, die Performance und Effizienz verschiedener Algorithmen zu vergleichen und zu analysieren. Die Implementierung von Algorithmen wie Insertion-Sort und Selection-Sort, die für den praktischen Einsatz bei großen Datensätzen weniger geeignet sind, soll vor allem zeigen, wie sich verschiedene Faktoren auf die Laufzeiten auswirken. Dadurch wird verdeutlicht, dass die Wahl des richtigen Algorithmus je nach Anwendung und Datensatzgröße entscheidend ist. Das Projekt konzentriert sich daher auf den theoretischen Vergleich und die Analyse der Algorithmen und nicht auf die tatsächliche Usability in einer realen Anwendung. Ein weiteres Ziel ist es, aufzuzeigen, wie bestimmte Änderungen für ein voll funktionsfähiges Deployment vorgenommen werden könnten, wenn die Applikation für reale Nutzer angepasst werden sollte.

4. Fazit und eigene Gedanken

Das Projekt war im Gesamten sehr erfolgreich: Alle grundlegenden Anforderungen wurden erfüllt, und darüber hinaus wurden eigene Ideen wie das Erstellen und Verwalten personalisierter Playlists erfolgreich integriert. Die Applikation bietet dem Nutzer ausreichend Feedback, um klar zu kommunizieren, welche Aktionen zulässig, erfolgreich oder fehlgeschlagen sind, was die Benutzererfahrung erheblich verbessert.

Die Implementierung der verschiedenen Algorithmen war technisch zwar unkompliziert, aber die Einbindung in eine funktionale, benutzerfreundliche Anwendung erwies sich als komplexer. Besonders die Notwendigkeit, ungültige Benutzereingaben zu verarbeiten, hat die Entwicklung herausfordernd gemacht stellte aber eine Voraussetzung dar, um die Robustheit der Applikation zu garantieren.

Tests haben die Stärken und Schwächen der verschiedenen Algorithmen klar aufgezeigt, was als solide Entscheidungsgrundlage für zukünftige Optimierungen und Erweiterungen des Projekts dienen kann. Sollten zukünftige Iterationen geplant sein, können die in Kapitel 3 vorgeschlagenen Optimierungen wie die Erweiterung der Datenquelle und die Fokussierung auf performantere Algorithmen wie Quick-Sort und binäre Suche den einer noch effizienteren und benutzerfokussierte Anwendung ermöglichen.