UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

LUCRARE DE LICENŢĂ

DJ Setlist Generator

Conducător științific
Lector Universitar, Dr. Mihoc Tudor

Absolvent
Mureșan Vlad

2021

ABSTRACT

Within this project, I have taken into consideration the next hypothesis: DJs choose the tracks which they are going to play in a certain order because they are trying to follow a tendency which describes how their setlist would feel. The ups and downs of their performance are considered, consciously or unconsciously, when choosing the tracks. Therefore, DJs have a trend when it comes to what track to play at what part of their setlist. Given this hypothesis, I have aimed to build a program which should be able to generate setlists of a given number of tracks, based on said trend, using collected data about popular DJ's performances and a genetic algorithm to properly generate the expected results.

Firstly, I have written a short introduction to outline the problem in detail, so as to achieve full understanding of what we want to attain.

In the second chapter, which is purely focused on the theoretical part of the problem, I have written briefly about what genetic algorithms are, just so it is possible for us to even talk about a custom algorithm built for our needs. After that, I have written about the parameters of our problem, more specifically, what we understand when we say how songs and setlists "feel", where we collect values from in order to build mathematical functions for our genetic algorithm and what they represent. Following this subchapter comes the part about our personal and custom genetic algorithm: what our chromosome represents, how we will evaluate a solution's correctness, what we expect our generated setlists to imitate and its overall desired performance.

The third chapter is describing the program itself. Starting from drawing out a list of tasks which need to be successfully completed in order to get on working on the next one, it follows into writing about each of these tasks individually. Firstly, a short section about the programming language and technologies used in this project. Then, a part about collecting all the data we need before building the genetic algorithm, the methods used and reasoning behind it. After this, there is a section about obtaining the mathematical functions that would describe the way setlists "feel", functions which our genetic algorithm will have to imitate. Finally, the part about the algorithm itself, the fitness function and how it was made, testing various operators in order to reach desired performance and some observations made after finishing the algorithm. The chapter ends with the graphical interface that was made in order to make the usage of this program easier.

The fourth chapter is about validating the results both theoretically and practically, and the fifth and final chapter is about different ways to extend the project in the future, and a short conclusion.

Cuprins

Capitolul 1

Introducere

De-a lungul istoriei, muzica a fost un element important din viața oamenilor. Putem spune că este un aspect al existenței ce ne unește pe toți, indiferent de origini. Odată cu modernizarea și schimbarea rapidă a tuturor domeniilor de activitate, industria muzicală a devenit un colos al economiei mondiale, iar rezultatul a fost apariția mai multor vocații ce au legătură cu aceasta, una dintre ele fiind *DJ*-ul. Acesta are rolul de a întreține energia și angajamentul unei adunări de oameni prin piesele pe care le pune și schimbările dintre acestea. O abilitate importantă a *DJ*-ului, pe lângă tehnica de care dă dovadă atunci când operează aparatul făcut special pentru meseria sa, este aceea de a selecta piesele potrivite. Acest proces este îndeplinit aproape întotdeauna înaintea spectacolului, iar el, în funcție de numărul de piese pe care trebuie să le pregătească, poate dura câteva ore. Oare nu poate fi automatizat acest proces? Acele ore pe care *DJ*-ul le consumă pentru a pregăti piesele, pot fi reduse la câteva minute prin intermediul unui algoritm ce la finalul rulării să îi ofere lista de piese în ordinea pe care *DJ*-ul să le pună la spectacol. În cadrul acestei lucrări, se va prezenta un astfel de program, abordarea problemei, conceptele matematice și informatice folosite, precum și metodologia de lucru.

1.1 Ipoteză

Înainte de a realiza un program pe care să îl folosească un *DJ* în scopul de a primi o listă de piese în ordinea în care să le puna, luăm în considerare următoarea ipoteză care va dicta în totalitate cum va fi realizat acest program: *DJ*-ii atunci când își selectează piesele, urmăresc, conștient sau inconștient, o tendință comună. Putem fi de acord că noi, ca ascultători, simțim fiecare piesă în mod diferit. *DJ*-ii cu multa experiență stiu ce simt oamenii când ascultă anumite piese, așa că în clipa când își formează *setlist*-ul (lista de piese) iau în calcul dinamica publicului. Spre exemplu, un *DJ* bun nu își vă pregăti piese ce sunt din ce în ce mai energice, deoarece acesta

știe că își va obosi publicul prea repede. În schimb, va lua în calcul anumite clipe de respiro prin piese mai puțin active, tocmai pentru a-și ține urmăritorii atenți și captivați pentru adevăratele momente puternice. Dacă am putea cuantifica această tendință și desena un grafic, un *setlist* ar fi în mod evident cu urcușuri și coborâșuri. Așadar, selecția de piese nu este una întâmplătoare, ci se realizează prin anumite reguli subtile, făcând din ceea ce pare inițial ceva banal, un proces elegant și extrem de important pentru succesul spectacolului, iar *DJ*-ii mari și de succes urmăresc, în mare parte, o tendință comuna când îndeplinesc acest proces.

1.2 Scopul Lucrarii

Cu cele spuse în secțiunea anterioare, putem concretiza scopul final al lucrării: Realizarea unui program ce va genera setlisturi, urmărind acea tendință a *DJ*-ilor mari. Pentru atingerea acestui scop, trebuie să luăm în calcul, pe rând, diferitele procese care sunt necesar a fi îndeplinite:

1. Colectarea de date despre *setlist*-uri deja alcătuite și efectuate de către *DJ* mari și profesioniști;
2. Găsirea unei modalități de cuantificare a tendinței comune;
3. Alcătuirea unui algoritm genetic ce va genera setlisturile, bazandu-se pe tendința calculată;s
4. Realizarea finală a unei aplicații de tip desktop cu arhitectură stratificată ce va folosi algoritmul alcătuit anterior.

Capitolul 2

Teoria Necesară

În cadrul primului capitol, voi reda partea teoretică ce trebuie luată în considerare atunci când ne dorim a rezolva problema propusă, de la concepte generale până la raționamente subiective necesare pentru relatarea corectă a ipotezei.

2.1 Înțelegerea algoritmilor genetici

Folosirea principiilor evolutive definite de Charles Darwin pentru rezolvarea automată a problemelor a început în anii 1950 – 1960, datorită unor savanți ce au abordat interpetari diferite, în mod independent, în locuri diferite ale lumii. "Evolutionsstrategie " a fost ideea introdusă de Rechenberg, iar mai apoi preluată de Schwefel. Alții, pe numele de Fogel, Walsh și Owens au descris tehnica numită "Evolutionary Programming", unde potențialii candidați pentru o soluție erau reprezentați că automate finite. În schimb, tipul de algoritm evolutiv ce va fi folosit în cadrul acestei lucrări, este cel genetic, unde se va evalua o populație de soluții-canditat folosind operatori inspirați din variația și selecția naturala. [8]

Pentru a putea discuta în continuare despre algoritmii genetici și utilizarea acestora în aflarea unor soluții pentru problema dată, trebuie să definim terminlogia ce va fi folosită. Când ne referim la o mulțime de soluții potențiale, o numim "populație"; în cazul unei singure soluții o numim "individ". Codarea unei soluții potențiale este un "cromozom" , o parte a codării este o "genă", iar calitatea soluției este numită "fitness", cât de bine acea soluție rezolvă problema dată.

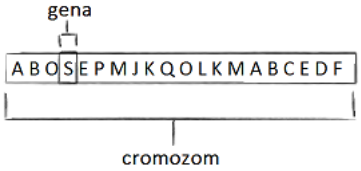

Figura 2.1: Exemplu de cromozom

Structura de bază a unui algoritm genetic este următoarea: inițializăm populația, verificăm calitatea soluției pentru cromozomi, aplicăm selecția, încrucișarea și mutația (operatori a căror scop este de a ne furniza cromozomi noi) și repetăm procesul, înafară de inițializare, până când găsim soluții ce respectă criteriul de oprire. Mai există un operator introdus de John H. Holland, numit inversiunea, dar care este rareori folosit în implementările de astăzi, iar avantajele sale nu sunt bine stabilite.

Odată ce avem structura, trebuie concretizată proiectarea unui algoritm de acest tip: Întâi, alegem o reprezentare a cromozomilor. Deseori, aceștia sunt sub forma unor secvențe de biți, numere, sau caractere. Apoi urmează alegerea modelului de populație și stabilirea funcției de fitness (evaluare), aceasta funcție fiind defapt măduvă algoritmului, deoarece ea ne vă putea dicta dacă o soluție este bună. Pe urmă, stabilim operatorii genetici, selecția, încrucișarea și mutația, iar la final, stabilim criteriul de stop. În mod normal, algoritmul se vă opri în cazul în care nu se observa schimbări semnificative în calitatea cromozomilor de la o iterație la alta, sau dacă se atinge numărul dorit de iterații (generații). [6]

Selecția, încrucișarea și mutația – Cei 3 operatori importanți din procesul descris. Selecția poate avea scopuri diferite, cum ar fi selecția unor părinți pentru reproducere, sau selecția supraviețuitorilor pentru o generație nouă. Poate fi deterministă (cel mai "fit" este selectat) sau stocastică (cel mai "fit" are cele mai mari șanse de a fi selectat). În cazul selecției pentru reproducere, poate fi făcută spre exemplu bazată pe fitness, sau prin turnir, iar în cazul selecției pentru supraviețuire, ea poate fi bazată pe fitness sau pe vârstă. Mutația este responsabilă cu schimbarea genelor în cromozomi. Scopul acesteia este de a reintroduce material genetic pierdut și de a introduce diversitate în populație. Exista multe tipuri de mutații, în funcție de tipul de reprezentare a cromozomului:

Reprezentare	Mutatie
binară	tare, slabă
întreagă	creep, random resetting
permutare	prin interchimbare, prin inserţie, prin inversare, prin amestec, k-opt
reală	uniformă, neuniformă: Gaussiană, Cauchy, Laplace
arborescentă	grow, shrink, switch, cycle, tip Koza

Figura 2.2: Tipurile diferite de mutații

Încrucișarea este operatorul ce are ca scop amestecarea informațiilor de la părinți pentru a genera un descendent. Aceasta poate fi făcută cu punct de tăietură, uniformă, sau prin alte metode ce din nou diferă în funcție de reprezentarea cromozomului.

Avem de-a face cu două tipuri de algoritmi genetici: *Generational* și *Steady-State*. În cazul algoritmului *generational*, avem o parte mare a populației ce va fi selectată pentru reproducere, urmașii lor fiind supuși mutației și introduși în populație, inlocuindu-i pe cei vechi, formând o generație nouă. În cazul celui *Steady-State*, selectăm doi cromozomi pentru reproducere, alegem un cromozom să "moară", îl înlocuim cu cel nou și repetăm procesul. Se poate observa că în al doilea caz, nu putem discuta de mai multe generații, ci doar de una asupra căruia lucram. Din punct de vedere computațional, rezolvarea *generational* are costul mai ridicat.

1. Initializeaza populația P la întâmplare
2. Evaluează fitness-ul întregii populații
3. Initializeaza populația P' goală de mărimea populației P
4. Selectează doi părinți din P pentru a crea copilul C prin
încrucișare (reproducere sexuală) , sau C ca fiind copie a unui părinte
selectat (reproducere asexuala)
5. Aplică mutația asupra lui C
6. Introdu-l pe C în P' și revino la pasul 4 până când P' e plin
7. Înlocuiește-l pe P cu P'
8. Revino la pasul 2 până când este îndeplinită condiția de stop

Figura 2.3: Algoritm Genetic *Generational* banal

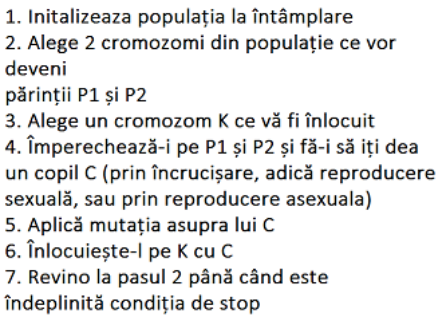

Figura 2.4: Algoritm Genetic *Steady-State* banal

Putem observa că exista posibilitatea de îmbunatățire a acestor algoritmi banali prin diferite optimizări. Pentru algoritmul *generational*, avem opțiunea de a garanta trecerea cromozomului cu fitness-ul cel mai mare către următoarea generație, sau în cazul celui *Steady-State*, alegerea părinților poate fi făcută prin selecție de tip turnir, luam 2 cromozomi și spunem că cel mai "fit" vă fi P1, iar pentru P2 procedăm la fel. [3]

În cazul problemei descrise în aceasta lucrare, deocamdată trebuie să ne definim cromozomul, să descriem ce înseamnă o soluție buna, să alegem criteriul de evaluare și să stabilim punctul de oprire.

2.2 Definirea tuturor parametrilor

Pentru a putea descrie algoritmul genetic ce va fi alcătuit pentru a rezolva problema propusă, trebuie să definim ce înseamnă un *setlist*, care sunt acele caracteristici pe care dorim să le urmărim. Mai exact, cum cuantificam cum se simte o piesă? Dar cum se simte un *setlist* întreg? Care este diferența dintre un *setlist* de 10 piese și unul de 50 de piese? Pentru început, trebuie să concretizam cum putem afla valori pentru caracteristici importante ale unei piese ce într-un final exprimă cum simțim noi acea piesă. Din fericire, avem acces la API-ul celor de la Spotify.

Spotify este o aplicație/serviciu de streaming pentru muzică și podcast-uri. În cadrul său, există opțiunea de a asculta un playlist bazat pe o piesă selectată, unde celelalte piese sunt asemănătoare celei inițiale. De asemenea, dacă noi căutam o piesă și o ascultăm, când e gata, vom primi automat alta ce inițial pare a fi dată aleator, însă ea este, din nou, asemănătoare celei căutate. Aceste fapte ne pot face să credem că există un algoritm foarte inteligent în spate ce învață pe parcurs ce piese ascultă oamenii una după alta, iar apoi reușește să îți sugereze piese ce ar fi pe placul tău. Acest lucru este adevărat și nu doar atât, Spotify a reușit printr-un alt algoritm

să cuantifice atribute abstracte ale pieselor. Când consultam documentația lor, putem vedea că fiecare piesă ce poate fi ascultata pe platforma lor conține următoarele caracteristici:


```
{
  "danceability": 0.92,
  "energy": 0.654,
  "key": 11,
  "loudness": -3.051,
  "mode": 0,
  "speechiness": 0.0401,
  "acousticness": 0.0236,
  "instrumentalness": 0.0158,
  "liveness": 0.0359,
  "valence": 0.847,
  "tempo": 117.046,
  "type": "audio_features",
  "id": "5ChkMS8OtdzJeqyybCc9R5",
  "uri": "spotify:track:5ChkMS8OtdzJeqyybCc9R5",
  "track_href":
"https://api.spotify.com/v1/tracks/5ChkMS8OtdzJeqyybCc9
R5",
  "analysis_url": "https://api.spotify.com/v1/audio-
analysis/5ChkMS8OtdzJeqyybCc9R5",
  "duration_ms": 293827,
  "time_signature": 4
}
```


Figura 2.5: Caracteristicile pentru piesa ”Michael Jackson - Billie Jean”

Putem observa niște atribute interesante, ce vor fi exact atributele ce le vom folosi pentru algoritmul genetic: ”danceability” (cât de dansabila este piesa), ”energy” (energia piesei) , ”valence” (cât de fericită este piesa). Aceste trei vor forma, de fapt, trei funcții matematice diferite, iar scorul de fitness va fi compus din toate trei rezultatele. [1]

Se pare că putem concepe un Data Pipeline ce va culege date despre setlisturi compuse de *DJ* mari de pe un website ce le documentează, și le salvează în baza noastră de date. Apoi, folosind API-ul de la Spotify, căutam fiecare piesa pe rand, colectăm valorile caracteristicilor și construim funcțiile matematice. Procedeul prin care vom construi funcțiile va fi discutat în capitolul practic, unde voi descrie diferitele metodologii ce pot fi folosite și raționamentul alegerii finale.

O mențiune importantă este faptul că *setlist*-urile ce vor fi culese au, în mod evident, număr diferit de piese de la unul la altul. Acest fapt nu este descurajator, deoarece pe noi ne interesează comportamentul lor și alegerea lor în diferite etape ale setului, nu câte ar fi. Din această cauză, domeniul funcției va fi [0,1] pentru toate piesele. X-ul se va calcula prin următoarea formulă:

$i * 1/n$ unde "i" este poziția piesei în *setlist*, iar "n" este numărul total de piese.

Acest raționament ne dă posibilitatea de a ne putea folosi de *setlist*-urile pentru care nu avem valorile căutate la toate piesele componente (adică marea marea majoritate) din diferite motive: piesele nu se află pe Spotify, piesele nu sunt documentate pe site-ul de unde luăm setlisturile, piesele nu sunt încă lansate etc. Pentru aceste setlisturi, unde avem "găuri" , nu putem trage concluzii despre trend-ul întregului set, dar putem trage concluzii despre părțile pe care le cunoaștem, deoarece ele reprezintă părți din set. Spre exemplu , dacă avem un *setlist* pentru care cunoaștem piesele și să spunem valoarea energiei lor până la jumătatea setului, iar cea de-a doua jumătate ne este complet necunoscută, nu putem înțelege trend-ul primei jumătăți? Ba da, iar acest fapt îl putem generaliza folosindu-ne de re-domenizarea setului intre 0 și 1.

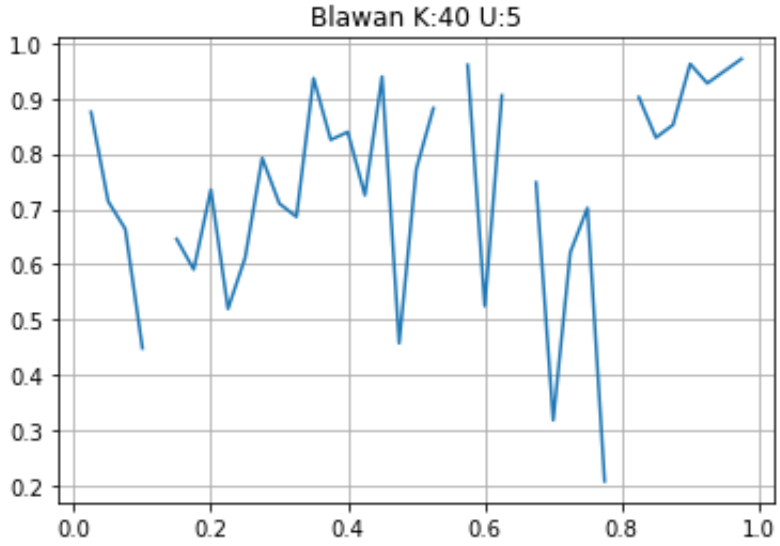

Figura 2.6: Trend-ul pentru "energy" a unui *setlist* de "Blawan"

Aici avem un grafic ce reprezintă un *setlist* prestat de Blawan, cu 40 de piese totale și 5 necunoscute. Pe axa Ox avem poziția piesei în *setlist* restrânsă în [0,1], iar pe Oy valoarea atributului "energy" luat de pe Spotify. Cu toate că se observa niste gauri, putem înțelege (evident, într-un mod superficial deocamdată) trend-ul energiei în acest *setlist*. O urcare treptată până la jumătate, iar apoi coborâri bruște , totul culminând la final într-un *all-time-high* al energiei. Nu este acest set valoros pentru construirea funcției finale? Întocmai că este, fiindcă putem trata bucățile unde cunoaștem datele ca fiind părți indivuduale ce reprezintă un anumit moment într-un *setlist*.

2.3 Aspectele algoritmului genetic al problemei date

Am precizat că scopul este acela de a avea un program ce poate genera unui *DJ* un *setlist* pe care acesta să-l presteze. Un *setlist* înseamnă o succesiune de piese muzicale, ordinea fiind la fel de importantă precum selectarea pieselor. De aici putem deduce că în acest caz, cromozomul va fi chiar un *setlist*: un șir de piese, iar piesa spunem că va conține: numele piesei împreună cu artistul într-un string și trei valoari reale ce vor descrie caracteristicile dorite ale piesei (energy, danceability, valence).

Lungimea cromozomului va fi, evident, numărul de piese dorite pentru *setlist*-ul generat. În calcul se va lua doar valoarea caracteristicii dorite: Cromozomul nostru va conține valori reale. Datorită naturii problemei, nu putem alege din prima un anume tip de încrucișare sau mutație, ci operatorii folosiți vor trebui testați și schimbați astfel încât să găsim combinația potrivita pentru a realiza un algoritm performant. Dar întâi, trebuie să definim concret și clar ce va însemna un rezultat bun, ce va însemna un cromozom "fit".

Considerăm o potențială soluție un cromozom de lungime "n", iar atomul cum a fost caracterizat anterior (o valoare reală ce descrie o caracteristică specifică a piesei). De asemenea, considerăm că avem o funcție bidimensionala a aceleiași caracteristici, ce descrie tendință generală pe care vrem să o reproducem. De dragul exemplului, caracteristica luată va fi "energy".

Fie $f(x)$ funcția calculată pe care vrem să o reproducem, $g(x)$ funcția pe valorile setlistului generat.

Avem următoarele variante pentru metoda de evaluare a soluțiilor:

2.3.1 Diferența Integralelor

Știm că valoarea integralei unei funcții definește aria de sub graficul funcției. Folosindu-ne de asta, putem concepe o formulă pentru a calcula aria dintre două grafice. [4] In cazul nostru, aceasta este :

$$|\int_0^1 f(x)\,dx - \int_0^1 g(x)\,dx|$$

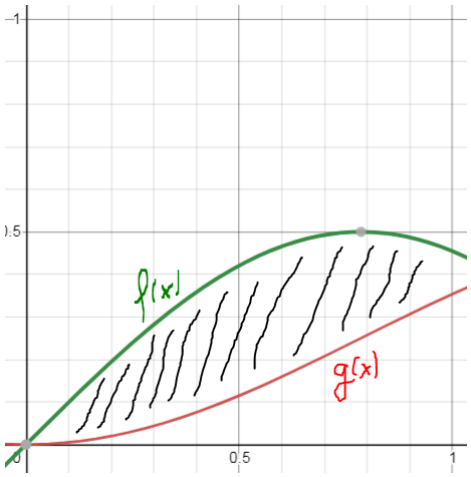

Figura 2.7: Exemplu

Cu cât această valoare se apropie mai tare de 0, cu atât funcția *setlist*-ului generat este mai aproape de cea după care vrem să ne luăm. Această soluție este una simplă, întrucât nu necesită o implementare foarte grea. La prima vedere pare a fi o validare corectă, însă putem observa că rezultatul va da 0 și în cazul în care cele doua funcții sunt oglindite, fapt ce nu ar fi corect. Vrem să urmărim trend-ul pe etape, iar într-un caz ca cel precizat, am avea o soluție fiind clasificata ca fiind bună, cu toate că ea ar ajunge să exprime un *setlist* total diferit de ceea ce ne dorim.

2.3.2 Diferența dintre Puncte și Funcție

Cromozomul nostru conține o listă de valori reale, nu neapărat o funcție. Nu este necesară construirea funcției pentru verificarea faptului că respectă funcția f, putem lua rând pe rând punctele și însumam diferența dintre ele și funcția f. Cu cât această suma este mai mică, cu atât soluția noastră este mai bună. Așadar, avem formula:

$$\sum_{k=1}^{n} |y_k - f(x_k)|$$

, unde y_k este valoarea caracteristicii piesei de pe poziția k (adică atomul de pe poziția k din cromozom) și x_k este poziția relativă a piesei între 0 și 1.

După cum spuneam, putem vedea că în cazul în care această sumă este mare, scorul pe care îl putem da soluției potențiale va fi mic, iar în cazul contrar, scorul va

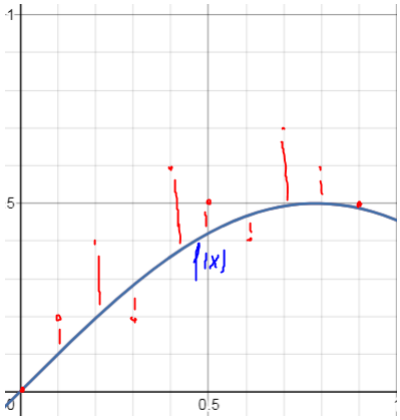

Figura 2.8: Exemplu

fi mai mare. Cu această validare, putem avea soluții ce teoretic ar trebui să se plieze bine pe funcția f. Dar în cazul nostru, noi nu vom avea o singură funcție pe care dorim să o imitam, ci trei, deoarece avem trei atribute pe care le-am ales că ar dicta cum se simte un *setlist*: ”energy”, ”danceability”, ”valence”. Ne putem da seama de pe acum că algoritmul nostru nu ar performa la un nivel foarte înalt din cauza că verifica trei funcții deodată. Spre exemplu, poate imita bine funcția ”energy” dar pe celelalte două deloc. De asemenea, chiar și în cazul în care reușim să obținem o soluție perfecta, aceasta nu ar fi decât o imitație a unei medii de funcții construite pentru un număr mare de *setlist*-uri. Aici intervine adevărata problemă de logică: Ce mai exact ne dorim să verificăm odată ce avem funcțiile medii pentru cele trei atribute? Dacă continuăm pe această variantă de validare, o soluție bună defapt nu ne-ar satisface așteptările, deoarece pe noi nu ne interesează ca piesele alese să aibă valorile atributelor apropiate de valorile medii, ci tendința de creștere sau coborâre a acestora în anumite momente cheie dintr-un *setlist*. Aici intervine cea de-a treia, și ultima, variantă pentru verificarea soluțiilor și acordarea unui scor de fitness.

2.3.3 Varianta Finală

Alegerea acestei variante, precum și conceperea ei, a venit odată cu analizarea întrebării ”Ce considerăm o soluție bună?”. Deja presupunem că avem funcția medie a caracteristicii ”energy” pe care dorim că *setlist*-ul generat să o imite, dar ce mai exact dorim să imite? Dacă doar primim un *setlist* a căror piese au exact valorile funcției în punctele respective, vom avea o soluție banală, din două motive:

- Deoarece funcțiile medii vor fi calculate pe mai multe *setlist*-uri, este destul de clar că valorile caracteristicilor vor fi într-un interval foarte mic. Acest fapt face că soluția dată să aibe piese cu diferențe mici de valori, ceea ce face să nu simțim cu adevărat schimbările;

- Nu ne dorim să avem chiar același rezultat de fiecare dată. Din cauza faptului că fiecare *setlist* este până la urma diferit, noi vrem să imităm anumite trend-uri din funcțiile medii calculate, nu să generam *setlist*-uri ce se simt la fel.

Ce înțelegem de fapt prin imitația unei funcții medii, într-o anumită măsură, este defapt imitația schimbărilor de valori de la un punct la următorul. Pentru a putea descrie concret procedeul prin care vom verifica acest fapt, includ o imagine ce reprezintă graficul funcției medii pentru atributul "energy" calculat din toate *setlist*-urile adunate, pe 100 de puncte echidistante.

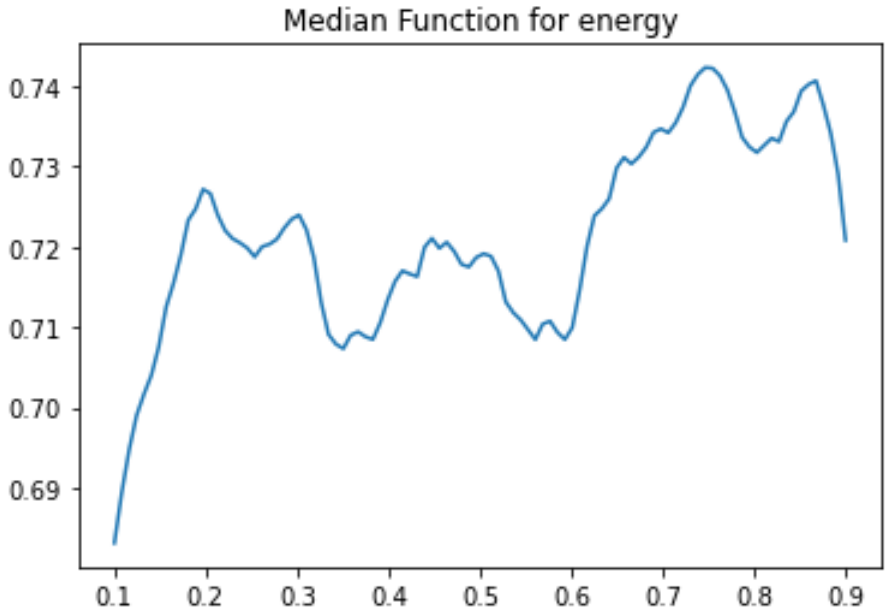

Figura 2.9: Funcția medie a energiei

După o analizare scurtă a acestui grafic, observăm că în general, un *setlist* trebuie să respecte următoarele principii: La început, să crească în energie, apoi să scadă puțin și să fluctueze pe la mijloc, iar în ultima treime a *setlist*-ului, să crească și mai mult. Din nou, vedem că valorile energiei se afla într-un interval mic, de aproximativ 0.05. Într-un *setlist* generat, putem avea un interval mai mare, așa că vom calcula următoarele lucruri:

1. $f_{diff} = max(f(x)) - min(f(x))$, pentru a afla mărimea intervalului codomeniului funcției calculate.

2. $g_{diff} = max(g) - min(g)$, unde g este cromozomul generat, pentru a afla mărimea intervalului în care se regăsesc valorile (finite) ale atributului ales (energy deocamdată) din cromozomul generat.

3. $rate = \frac{g_{diff}}{f_{diff}}$, pentru a afla rata de conversie pe care o vom folosi la pașii următori.

4. $best = rate * (f(x_{k+1}) - f(x_k), k = 0...n-1$, unde k ia valorile pozițiilor relative a *setlist*-ului generat.

5. $actual = y_{k+1} - y_k, k = 0...n-1$, unde y_k este valoarea caracteristicii energy (în exemplul nostru) pentru piesa de pe poziția relativa k a *setlist*-ului generat.

În momentul de față, $best$ conține cea mai bună valoare pentru diferența de la punctul actual la următorul, deoarece înmulțim diferența valorilor din funcția medie cu rata de conversie, astfel încât să avem același comportament indiferent de "mărime". În $actual$ avem adevărata diferență dintre punctul următor și cel curent. Acum avem nevoie de o metodă pentru acordarea unui scor în funcție de cât de diferite sunt aceste valori $best$ și $actual$.

Pentru început, putem spune direct că acordăm 0 în cazul în care $best$ și $actual$ au semne opuse. Acest fapt ne-ar spune că în soluția noastră, trendul o ia fix invers, că urcă atunci când ar trebui să coboare, sau invers.

În caz contrar, în care semnul este același, trebuie să verificam cât de mare este diferența dintre $best$ și $actual$. Avem nevoie de o funcție matematică pentru care să rezulte valoarea 1 în cazul în care aceste valori sunt egale și 0 în cazul în care diferența este prea mare. Pentru această etapă, am ales să folosesc o funcție de gradul 2 de următoarea formă:

$$(\frac{x}{best})^2 + 1$$

Graficul acestei funcții pentru $best = 0.3$ arată în felul următor:

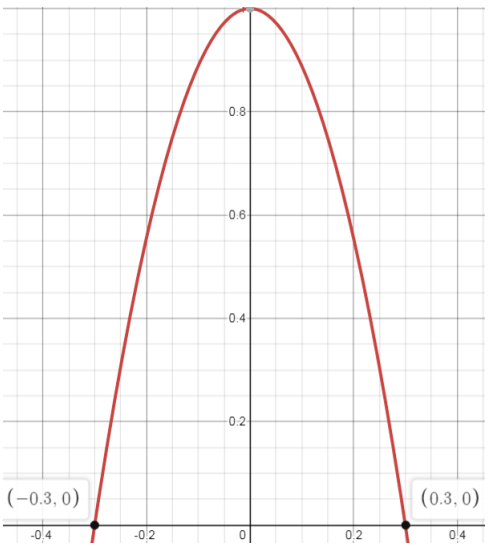

Figura 2.10:

Dacă înlocuim x cu valoarea scăderii $best - actual$ observăm că vom avea 1 în cazul în care ele sunt egale, 0 dacă $actual = 2 * best$ sau $actual = 0$ și valori pozitive între 0 și 1 pentru $actual \; \epsilon (0, 2 * best)$. Pentru valori dinafară acestui interval, vom acorda automat valoarea 0.

Așadar, pentru fiecare pereche de puncte (x_k, x_{k+1}) vom avea un scor ce poate lua valori de la 0 la 1. Adunăm scorul pentru aceste perechi, și într-un final îl împărțim la lungimea cromozomului, pentru a incadra și cel final între 0 și 1.

Repetăm acest proces și pentru celelalte două atribute importante, "danceability" și "valence", iar la final vom avea un scor de fitness rezultat din toate cele trei verificări. Această metodă finală va da un scor bun potențialelor soluții ce reușesc să facă ceea ce ne doream inițial și era doar o cerere abstractă: a imita într-o anumită măsură trend-ul unor funcții în anumite momente. Ne așteptăm ca după implementarea acestei metode, soluția generată în final de algoritm să aibe graficele atributelor asemănătoare graficelor funcțiilor medii calculate, întrucât să putem "arăta cu degetul" similaritățile rezultate.

2.3.4 Performanța Dorită

Motivul alegerii implementării unui algoritm genetic pentru rezolvarea problemei implică și natura să imprevizibilă, fapt ce ne poate garanta că de la o rulare la alta, soluțiile să fie total diferite. Pe de altă parte, putem garanta că algoritmul ne vă genera *setlist*-uri ce nu numai că respectă regula impusă, dar și vor conține anumite segmente neașteptate, în cazul în care în implementarea să nu vom avea în gând să ajungem la un scor de 100%, ci mai degrabă unul de 75%-80%. Acest lucru este dorit, deoarece problema pusa are un anumit indice de subiectivitate, încât da, ne dorim să urmărească trend-urile calculate, dar și să avem câteva segmente ce ies din tipar. Prin urmare, o proporție de 1/5-1/4 din *setlist* să fie nu neapărat conform regulilor, este chiar un lucru dorit.

2.3.5 Parametrii Algoritmului

Se va folosi un algoritm de tip *generational*, iar odată cu implementarea acestuia împreună cu operatorii săi, se va hotărî (în partea practică a lucrării) numărul de generații pentru care să ruleze algoritmul și mărimea populației. Valorile finale vor fi date după câteva teste ce vor avea în vedere performanța dar și păstrarea unui timp scurt de rulare.

Capitolul 3

Lucrarea Practică

În acest capitol voi descrie și arăta procedeele practice folosite pentru realizarea problemei date, precum și raționamentul ce a condus spre alegerea lor. Am să precizez etapele parcurse, provocările ce au apărut, dar și momentele cheie ce au făcut ca scopul să fie, într-un final, îndeplinit.

3.1 Modus Operandi

Odată ce am definit și descris aspectele teoretice ale problemei, trebuie stabilit un plan de lucru, o succesiune de etape ce vor conduce într-un final spre realizarea proiectului.

1. Pregătirea datelor
    (a) Definirea domeniului și întocmirea bazei de date;
    (b) Crearea unui webscraper ce va accesa site-ul "mixesdb.com" pentru a culege date despre setlisturi;
    (c) Rularea webscraperului astfel încât să avem un număr semnificativ de setlisturi în baza de date;
    (d) Crearea unui algoritm ce va accesa API-ul de la Spotify pentru a strânge toate datele pieselor ce se afla în setlisturile salvate (acolo unde se pot găsi);
    (e) Salvarea unui dataset de pe kaggle sau alt website ce conține atribute de piese de pe spotify pentru a avea de unde alege algoritmul genetic.
2. Algoritmul genetic
    (a) Găsirea unei modalități prin care putem construi funcțiile matematice ce vor fi folosite în calcularea corectitudinii setlistului generat;

(b) Întocmirea algoritmului genetic;

(c) Optimizarea algoritmului acolo unde este nevoie.

3. Starea Finală

(a) Crearea unui GUI pentru folosirea programului.

(b) Conversia în .exe astfel încât să nu fie necesar un IDE pentru rulare.

4. Validare

(a) Verificarea rezultatelor prin testarea propriu zisă a setlist-ului.

3.2 Tehnologiile alese

Înaintea realizării proiectului propus s-au decis tehnologiile care urmau a fi folosite. În cadrul acestui subcapitol, am să precizez fiecare aspect în parte.

3.2.1 Limbaj de Programare & IDE, Baza de Date

Deoarece problema dată va necesita lucrul cu date, alcătuirea unui script pentru colectarea acestora și construirea unui algoritm genetic, este nevoie de un limbaj puternic pe care ne putem baza. Alegerea a fost Python, iar IDE-ul folosit a fost Spyder de pe distribuția Anaconda. Motivele selectării acestui IDE au fost faptul că librăriile necesare pentru îndeplinirea unor sarcini erau deja incluse, debug-ul este foarte detaliat, execuție interactivă și inspecție cuprinzătoare.

Pentru baza de date s-a folosit PostgreSQL din motivul lipsei de familiaritate cu acest sistem. Datorită faptului că nu a fost nevoie de foarte multă modelare, am folosit shell-ul inclus.

3.2.2 Librăriile Adiționale

Cu toate că Spyder include toate librăriile importante pentru domeniul ”Data Science”, precum ”numpy”, ”scipy”, ”matplotlib” (care au fost folosite), avem nevoie în plus de alte librarii. Pentru lucrul cu baza de date, am avut nevoie de ”SQLAlchemy” și ”psycopg2”. Pentru navigarea site-ului de setlisturi și colectarea datelor, am folosit ”BeautifulSoup4”, iar pentru folosirea ușoară a API-ului de la Spotify, am folosit o librărie numită ”spotipy”.

3.3 Pregătirea Datelor

În problema noastră, avem nevoie de mai multe date din diferite locuri:

1. *setlist*-uri de pe website-ul mixesdb.com;
2. Atributele pieselor din setlisturile salvate, cu ajutorul API-ul de la Spotify;
3. Un set mare de date compus din piese cu atributele de pe Spotify, din care algoritmul genetic să aleagă.

3.3.1 Webscraping

Pentru prima etapă a fost necesară construirea unui script de webscraping care să acceseze website-ul menționat și să culeagă datele necesare nouă. Scriptul trebuia să opereze în următorul fel:

1. Să acceseze pagina ”Hottest mixes” de site;
2. Să intre pe fiecare *setlist* în parte;
3. Să salveze numele *DJ*-ului căruia îi aparține *setlist*-ul, anul în care a fost efectuat și fiecare piesă în ordine;
4. Să acceseze pagina următoare;
5. Să repete pașii 2-4 până când se ajunge la numărul de pagini cerute.

Pentru a putea scrie codul unui astfel de script, trebuie să inspectăm codul html corespunzător. Pagina ”Hottest Mixes” conține o listă de 100 de redirect-uri către pagini pentru *setlist*-urile respective precum și un buton care ne duce către următoarea pagină ce conține alte 100 de aceste redirect-uri. Odată ce inspectăm codul, putem vedea că fiecare redirect este un tag ”a” cu un ”href” care indică url-ul setlistului. Singurele tag-uri ”a” cu ”href” precizat sunt aceste redirecturi către setlisturi, precum și acela ce ne duce pe pagina următoare, așa că vom colecta toate linkurile într-o listă pentru a le accesa ulterior, iar cel pentru ”next page” îl vom ține minte separat.

Pagina ”Hottest Mixes” o vom da ca parametru, așa că primul pas este realizat. Al doilea este un lucru ușor de efectuat datorită clarității codului html al paginii. În continuare avem metoda din program responsabilă pentru acest pas, împreună cu metoda ce abstractizează pagina într-un obiect BeautifulSoup. [9]

```
def get_html_of_url(self,url):
        from urllib.request import urlopen
        from bs4 import BeautifulSoup
        page = urlopen(url)
        html_bytes = page.read()
        html = html_bytes.decode("utf-8")
        soup = BeautifulSoup(html,"html.parser")
        return soup

 def get_setlist_links_and_next_page(self,url):
        soup = self.get_html_of_url(url)
        mixesLinks=[]
        nextPageLink=""
        for a in soup.find_all('a',href=True):
            posLink = str(a['href'])
            if (len(a.contents)>0):
                if (str(a.contents[0]) =='next 100'):
                    nextPageLink = "https:" + posLink
            if (posLink.startswith(("/w/2","/w/1"))):
                mixesLinks.append("https://www.mixesdb.com"+posLink)

        return mixesLinks,nextPageLink
```

Pentru pașii 3 și 4 avem nevoie de o metodă (1) care colectează datele având o pagină url primită că parametru, iar alta să conțină cea scrisă mai sus și să apeleze (1) pentru fiecare *setlist*. Pentru metoda (1) avem nevoie din nou să inspectam codul html al paginii unui *setlist*. La prima vedere, observăm că tracklist-ul este scris piesă cu piesă în tag-uri ”li” sub un heading ”h2”, dar la prima rulare a scriptului, am constatat că am salvat mai puține *setlist*-uri decât mă așteptam. Când am inclus o secvență de cod astfel încât să îmi semnaleze dacă pentru o pagină nu îmi găsește structura html așteptată, am dedus că mai există un fel în care este ”formatată” pagina. În loc de ”li”-uri, aveam ”div”-uri cu class ”list-track”, dar și alte pagini destul de rare care păreau să nu respecte nicio regulă, așa că nu le-am luat în calcul. Într-un final, am rămas la aceste două cazuri.

```
def access_setlist_link(self,url):
        import re
        soup = self.get_html_of_url(url)
        tracks=[]
        artist=""
        date=""

        #FIND AND GET DATE
        div = soup.find("div",{"id": "mw-normal-catlinks"})
        ul = div.find("ul")
```

```
        date_li = ul.find_all("li")[0]
        date = str(date_li.text)

        #FIND AND GET ARTIST
        artist_li = ul.find_all("li")[1]
        artist = str(artist_li.text)

        #FIND AND GET TRACKS
        h2 = soup.find('h2',text='Tracklist')
        good_format=True;
        try:
            for sibling in h2.find_next_siblings():
                if (sibling.name == "h2"):
                    break;
                for li in sibling.find_all('li'):
                        track = self.remove_unnecessary_characters(str(li
    .contents[0]))
                        for child_a in li.find_all('a'):
                            track= track + self.
    remove_unnecessary_characters(child_a.contents[0])
                        #print(track)
                        tracks.append(track)
                if (len(tracks)==0):
                        for li in sibling.find_all('div',{"class":"list-
    track"}):
                            track = self.remove_unnecessary_characters(
    str(li.contents[0]))
                            #print(track)
                            tracks.append(track)
        except:
            good_format = False;
        extra_unknown=0
        for track in tracks:
            if not (len(track.split('-'))>1):
                extra_unknown+=1
        if (self.validate_setlist(tracks,extra_unknown)==False):
            return None,None,None

        return artist,date,tracks
```

Metoda ce apare în cod "remove_unnecessary_characters" nu face altceva decât să elimine caractere ce ne-ar încurca în căutarea pe Spotify, spre exemplu:
"**Roy Davis Jr - Gabrielle (Live Garage Mix)**" devine **"Roy Davis Jr - Gabrielle**"

Metoda "validate_setlist" verifică numărul de piese necunoscute dintr-un *setlist*, lucru ce se întâmplă destul de des. Piesele necunoscute sunt notate de obicei prin

”?”. Renunțam la un *setlist* dacă este prea mic, sau dacă nu cunoaștem mai mult de 66% din el. Apelăm metoda de mai sus în cadrul următoarei:

```
def access_pages(self,nr_of_pages):
        from domain.\textit{setlist} import \textit{setlist}
        import re
        total_tracks=0
        setlist_tracks_array=[]
        page=1
        mixesLinks,nextPage = self.get_setlist_links_and_next_page(self._initialUrl)
        for setlink in mixesLinks:
            current_tracks=[]
            current_artist = ""
            current_date = ""
            current_artist,current_date,current_tracks = self.access_setlist_link(setlink)
            print("We're on page ",page)
            if current_artist is not None:
                if re.match(r'^(19|20)\d{2}$',current_date) is None:
                    s = \textit{setlist}(0000,current_artist,current_tracks)
                else:
                    s = \textit{setlist}(current_date,current_artist,current_tracks)
                try:
                    self.__service.add(s)
                    total_tracks+=len(current_tracks)
                    print("\textit{setlist} by "+s.get_dj()+" added!")
                    print("Total tracks: "+ str(total_tracks))
                except:
                    print("Error!")

        for i in range(nr_of_pages): #100 setlists per page
            page+=1
            print("We're on page ",page)
            mixesLinks,nextPageTemp = self.get_setlist_links_and_next_page(nextPage)
            for setlink in mixesLinks:
                current_tracks=[]
                current_artist = ""
                current_date = ""
                current_artist,current_date,current_tracks = self.access_setlist_link(setlink)
                if current_artist is not None:
                    if re.match(r'^(19|20)\d{2}$',current_date) is None:
                        s = \textit{setlist}(0000,current_artist,
```

```
    current_tracks)
39                  else:
40                      s = \textit{setlist}(current_date,current_artist,
    current_tracks)
41                  try:
42                       self.__service.add(s)
43                       total_tracks+=len(current_tracks)
44                       print("\textit{setlist} by "+s.get_dj()+" added!
    ")
45                       print("Total tracks: "+ str(total_tracks))
46
47                  except:
48                       print("Error!")
49
50              nextPage= nextPageTemp
51          return setlist_tracks_array
```

Metoda de mai sus apelează toate de până acum, și îndeplinește toți pașii menționați. Prin intermediul unui service, salvează *setlist*-urile în baza de date, iar după apelarea acesteia pentru primele 10 pagini, datorita faptului că unele *setlist*-uri sunt prea mici, cunoaștem prea puține piese din ele, sau pur și simplu aveau un format care nu respectă nicio regulă, am ajuns să avem următoarea sumă de entry-uri salvate:

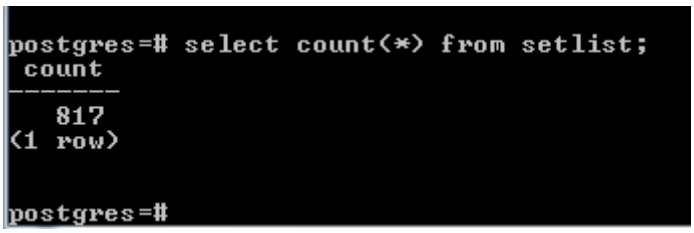

Figura 3.1:

3.3.2 Spotify API

Următorul pas spre colectarea tuturor datelor de care avem nevoie este să accesam API-ul Spotify pentru fiecare piesă din fiecare *setlist* și să le salvăm. Întâi, am creat o tabelă ”song” în baza de date ce conține toate caracteristicile unei piese pe Spotify ce ne-ar putea interesa, dar și atribute definite personal, cum ar fi ”search_string” ce semnifică string-ul folosit pentru căutarea piesei, sau ”relative_position” care este, după cum am mai precizat în capitolul teoretic, poziția piesei în *setlist* relativă la intervalul [0,1).

Apoi, asupra consultării documentației librăriei ”Spotipy”, am întocmit procedeul prin care vom putea căuta piesa pe spotify, iar în cazul în care o găsim, o salvăm cu valorile date ale caracteristicilor. În caz contrar, salvăm ”-999”, valoare

ce nu poate apărea natural la niciuna dintre atribute, și o vom ține minte pentru calculul funcțiilor ce va urma, încât să nu luam în considerare aceste valori. [7]

```
1
2     def search_song(self,song_text):
3         if not (len(song_text.split('-'))>1): #Verify song integrity (
      artist - name)
4             return None
5         song_text = song_text.replace('- ','')
6         try:
7             out = self.__spotify.search(song_text,type="track")['tracks'
      ]['items'][0]['id']
8         except:
9             return None
10            print("Didn't find "+ song_text)
11        return out
12
13    def get_song_features(self,song_text):
14        try:
15            rezultat = self.__spotify.audio_features(self.search_song(
      song_text))[0]
16        except TypeError:
17            return None
18        return rezultat
19
20    def assign_song_object_features(self,\textit{setlist},service):
21        from domain.song import Song
22        songs = \textit{setlist}.get_songs()
23        print(str(\textit{setlist}.get_date()) + " | " + \textit{setlist
      }.get_dj() + " | " + str(\textit{setlist}.get_id()))
24        for index,song_full_string in enumerate(songs):
25            splitted = song_full_string.split(' - ')
26            artist = splitted[0]
27            name = ' '.join(splitted[1:])
28            search_string = song_full_string.replace('- ','')
29
30            features = self.get_song_features(song_full_string)
31
32            try:
33                s = Song(\textit{setlist}.get_id(),song_full_string,
      search_string,artist,    name,index,len(songs),index*(1/len(songs)),
      features['acousticness'],
34                         features['danceability'],features['duration_ms'
      ],features['energy'],
35                         features['instrumentalness'],features['key'],
      features['tempo'],
36                         features['liveness'],features['loudness'])
```

```
37            except TypeError:
38                s = Song(\textit{setlist}.get_id(),song_full_string,
    search_string,artist,name,index,len(songs),index*(1/len(songs))
    ,-999,-999,-999,-999,-999,-999,-999,-999,-999)
39            service.add(s)
40            print(s.full_string + " added.")
41
42
43    def start_engine(self,setlist_array,service):
44        for \textit{setlist} in setlist_array:
45            self.assign_song_object_features(\textit{setlist},service)
```

După rularea codului, putem observa că avem 26092 de piese salvate, dar doar 16203 de piese cu valori reale, adică piese pe care le-am găsit pe Spotify, fie din cauză că nu erau din prima documentate pe site-ul de unde am cules *setlist*-urile, fie că piesa chiar nu se află pe platformă. Așadar, va fi nevoie de atenție sporită când vom construi funcțiile matematice pentru caracteristicile dorite.

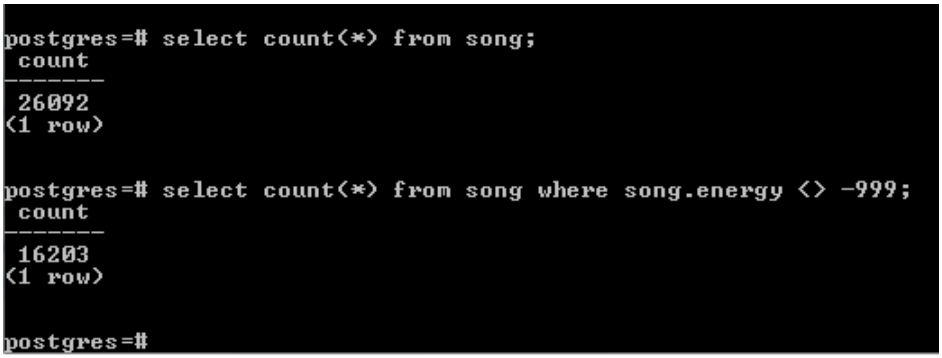

Figura 3.2:

3.3.3 Dataset-ul pentru Algoritmul Genetic

Într-un final, este nevoie de o mulțime de piese de unde algoritmul propus să aleagă atunci când generează *setlist*-uri. Pe Kaggle se găsesc o multitudine de fișiere tip "csv" ce conțin pe fiecare rând piese împreună cu atributele lor de pe Spotify, însă selecția domeniului este destul de importantă și trebuie făcută cu grijă din câteva motive:

- Deoarece algoritmul nostru va selecta piese în mod aleator, putem deduce faptul că un set cu piese foarte diferite intre ele ar duce algoritmul spre o performanta scăzută. Spre exemplu, un set cu piese a căror gen muzical se încadrează în vasta barcă numită "electronic", ar fi mult mai dorit decât unul ce conține piese de toate felurile, din toate colțurile și culturile muzicale ale lumii.

- *setlist*-urile luate pentru a înțelege trend-ul nu trec de domeniul electronic-ului.
- Chiar dacă am reușit să construim un algoritm extrem de performant, ce să respecte regulile propuse la perfecție și pentru piese total diferite între ele, am primi un rezultat eronat datorită faptului că în niciun context și sub nicio formă vreun *DJ* va avea în același *setlist*, spre exemplu, o piesă Hip-Hop românească și una tradițională tibetană.

Așadar, aceste observații ne conduc spre a alege un dataset compus din piese electronice. Din fericire, a fost găsit un fișier csv ce conține 21000 de piese, din următoarele *genre*-uri (specificate): techno, tech house, trance, psytrance, trap, hard style, drum and bass.

3.4 Obținerea Funcțiilor pentru Caracteristici

În clipa de față avem de-a face cu aproximativ 800 de *setlist*-uri și un total profund de 26000 de piese, din care doar pentru 16000 cunoaștem atributele. Problema pieselor lipsă se poate vedea în aproape toate *setlist*-urile salvate, dar datorită raționamentului transpunerii domeniului în intervalul [0,1), știm că ne interesează ce se întâmplă în anumite momente ale set-ului. Chiar dacă are 20 de piese sau 50, vrem să înțelegem cum se comportă la început, la final, la mijloc și așa mai departe. Așadar, faptul că există lipsuri în date nu este un lucru optim, dar nici un obstacol ce ar implica imposibilitatea realizării problemei. Aplicând un plot liniar primului *setlist* salvat pentru atributul "energy", unul realizat de către *DJ*-ul "Floating Points", acesta arată în felul următor:

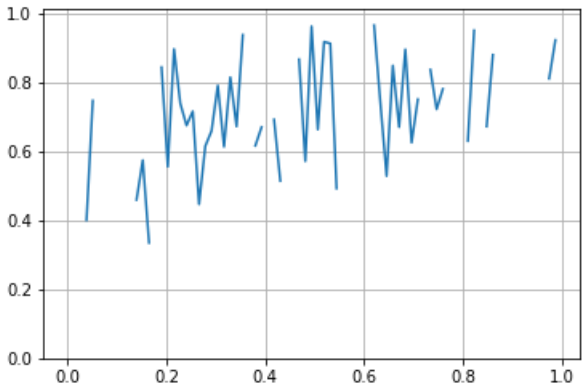

Figura 3.3: *setlist* Floating Points cu 79 piese, din care 24 sunt necunoscute

Dorim ca într-un final să avem câte o funcție matematică pentru fiecare din cele trei caracteristici alese (energy, danceability, valence), pentru a putea începe proiectarea algoritmului genetic. Acest lucru poate fi îndeplinit în mai multe feluri, dar

trebuie analizat fiecare și ales cel mai potrivit. Pentru a simplifica explicația, în continuare vom considera doar atributul ”energy”, iar pentru celalalte două vom avea aceeași soluție analog.

Un mod de abordare este construirea a câte o funcție pentru fiecare *setlist* salvat, iar la final calcularea unei funcții medii. Luând același *setlist* ca mai sus, putem încerca mai multe variante. Prima este construirea folosind metodele ”polyfit” și ”polyval” din numpy, ce ne vor rezulta un least squares fit pentru datele trimise. [5]

```
1  def poly_plot_setlist_trend(self,songs,desired_attribute,title):
2          import matplotlib.pyplot as plt
3          import numpy as np
4
5          x,y,unknown_songs=self.remove_nan_values(songs,desired_attribute)
6          y_vals = np.array(y)
7          x_vals = np.array(x)
8          pol = np.polyfit(x_vals,y_vals,len(x_vals)-1)
9          yy = np.polyval(pol,x_vals)
10
11         plt.plot(x_vals, yy, '-',x_vals, y_vals, 'ro')
12         title = title + " T:" + str(songs[0].setlist_length) + " U:" +
     str(unknown_songs)
13         plt.title(title)
14         plt.show()
```

Metoda de la linia 5 ”remove_nan_values” nu face altceva decât să verifice piesele pentru care nu cunoaștem caracteristica (are valoarea -999 după cum am hotărât când am colectat datele) și să returneze un array x ce conține lista pozițiilor relative la intervalul [0,1) și un array y ce conține valoarea caracteristicii date prin parametrul ”desired_attribute” pentru piesele cunoscute. Aplicarea procedeului *setlist*-ului ”Floating Points”, avem următorul rezultat:

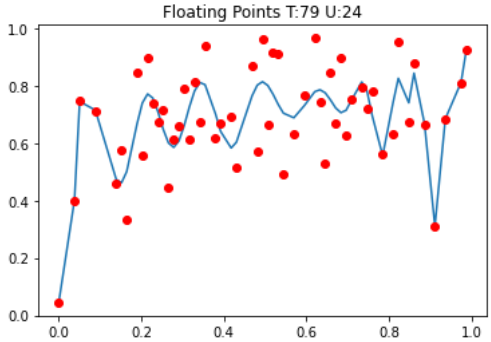

Figura 3.4: Floating Points, varianta polyfit, polyval

Putem observa consecința că funcția nu atinge toate punctele, iar acest procedeu

de ar fi aplicat pentru fiecare *setlist* în parte, nu ne-ar indica un rezultat corect pentru cele ce au un număr mai mare de piese. În schimb, pentru *setlist*-uri cu număr mai mic de piese, avem deznodământul așteptat. Iată un exemplu pentru un *setlist* compus de *DJ*-ul "Objekt" cu 27 de piese în total, din care 11 sunt necunoscute:

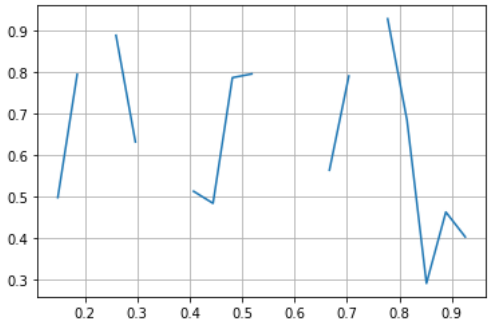

Figura 3.5: Objekt, reprezentare liniară

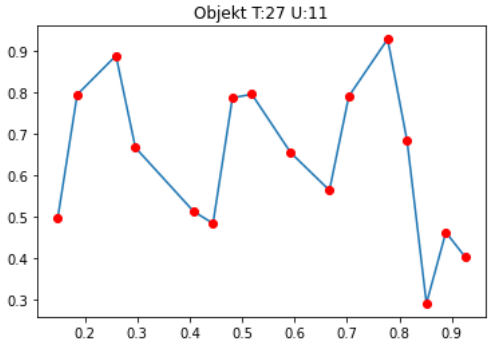

Figura 3.6: Objekt, varianta polyfit, polyval

În căutarea procedeului potrivit, o posibilă soluție a fost folosirea interpolării Lagrange, dar la consultarea documentației implementării acesteia din numpy, am dat peste următoarea precizare [2]:

```
'''
    Polynomial fits using double precision tend to "fail" at about
    (polynomial) degree 20. Fits using Chebyshev or Legendre series are
    generally better conditioned, but much can still depend on the
    distribution of the sample points and the smoothness of the data.  If
    the quality of the fit is inadequate, splines may be a good
    alternative.
'''
```

Din moment ce *setlist*-urile salvate aveau aproape toate peste 20 de piese și distante inegale între puncte, singura soluție viabilă era interpolarea spline.

```
    def spline_plot_setlist_trend(self,songs,desired_attribute,title):
        import numpy as np
        import matplotlib.pyplot as plt
        from scipy.interpolate import interp1d
        x,y,unknown_songs=self.remove_nan_values(songs,desired_attribute)
        x_vals = np.array(x)
        y_vals = np.array(y)

        f = interp1d(x_vals,y_vals,kind='cubic')
        xnew = np.linspace(x_vals[0],x_vals[len(x)-1],100)

        plt.plot(x,y,'x',xnew,f(xnew))
        title = title + " T:" + str(songs[0].setlist_length) + " U:" +
    str(unknown_songs)
        plt.title(title)
        plt.show()
```

Rularea metodei asupra *setlist*-ului de "Floating Points" ne rezultă următorul grafic:

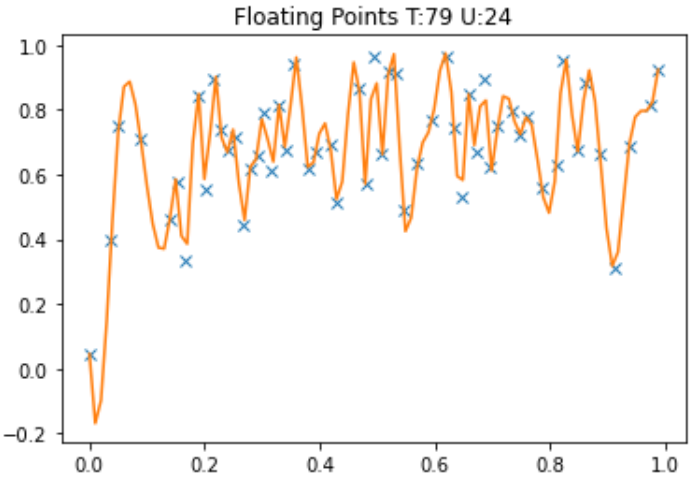

Figura 3.7: Floating Points, interpolare Spline

Din fericire, putem observa un comportament mult mai plăcut folosind această soluție, așa că vom calcula funcțiile medii prin acest procedeu. Realizarea acestui lucru a fost făcută în felul următor:

1. Construim funcție pentru fiecare *setlist* folosindu-ne de interpolarea spline;
2. Colectăm rezultatele funcțiilor în 100 de puncte echidistante între 0 și 1;
3. Calculăm media punctelor;
4. Construim funcția finală pentru media punctelor;

```
    def get_1d_interpolation(self,songs,desired_attribute):
        import numpy as np
        import matplotlib.pyplot as plt
        from scipy.interpolate import interp1d
        x,y,unknown_songs=self.remove_nan_values(songs,desired_attribute)
        x_vals = np.array(x)
        y_vals = np.array(y)
        try:
            f = interp1d(x_vals,y_vals,kind='cubic')
            return f
        except ValueError:
            return None

    def matrix_mean_calculation(self,functions,title):
        import numpy as np
        import matplotlib.pyplot as plt
        from scipy.interpolate import interp1d
        from scipy.integrate import quad
        x_all = np.linspace(0.1,0.9,100)
        f_all =[]
        for f in functions:
            this_f=[]
            for x in x_all:
                try:
                    this_f.append(f(x))
                except ValueError:
                    this_f.append(0)
            f_all.append(np.array(this_f))
        data_collection = np.vstack(f_all)
        f_avg = np.average(data_collection,axis=0,weights=data_collection
    .astype(bool))
        f = interp1d(x_all,f_avg,kind='cubic')
        plt.plot(x_all,f(x_all))
        plt.title(title)
        plt.show()
        return f
```

Prima metodă returnează funcția construită pentru un *setlist*, dar poate returna "None" în cazul în care apare o eroare cu *setlist*-ul dat. Spre exemplu, este posibil să nu fie găsită nicio piesă pe Spotify, iar atunci metoda să primească un array gol. La rulare, s-a observat că din 816 *setlist*-uri, au fost 810 ce au trecut cu succes, însemnând că 6 erau eronate.

A doua metodă este cea care primește o listă de funcții rezultate de la prima, iar pentru fiecare din ele colectează valorile rezultate pentru x de la 0.1 la 0.9 într-un numpy vstack, în cazul în care acestea nu depășesc domeniul de interpolare. De exemplu, pentru un *setlist* căruia îi cunoaștem doar a doua jumătate, nu îi pu-

tem cere funcției construite să ne dea valoarea în y pentru x = 0.3, în cazuri de genul acesta, adaugăm 0. Vstack-ul este o structură de tip matrice, care va avea aceste y rezultate pe fiecare rând. La final apelăm numpy average pentru a calcula media acestor rezultate, 0 nefiind luat în seamă datorită parametrului "weights=data_collection.astype(bool)". Motivul alegerii x-ului între 0.1 și 0.9, nu 0 și 0.99 (1 nu este atins oricum) este faptul că se observă că sunt mai puține valori de calculat la capetele intervalului, iar acest fapt produce calcule nepotrivite.

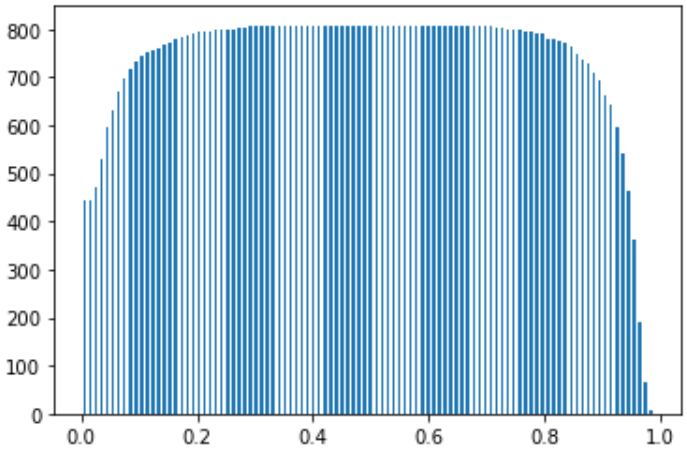

Figura 3.8: Histograma distribuției valorilor

Dacă restrângem în intervalul [0.1,0.9] nu putem considera că pierdem prea multe date. Într-adevăr, acest lucru se întâmplă din cauza faptului că după interpolari, nu se pot calcula corect valorile din capete, dar făcând acest "schimb", în care ajungem să tăiem funcțiile înainte de 0.1 și după 0.9, câștigăm formele unor funcții corecte și pe care le putem considera că atare, că 0.1 fiind noul început iar 0.9 noul final.

Cele două metode le apelăm în cadrul alteia pentru a duce la capăt procedeul.

```
def plot_median_interpolation(starting_id,final_id,desired_attribute):
        from utils.plotting import Plotting
        import numpy as np
        plotting = Plotting()
        title = 'Median Function for ' + desired_attribute
        functions = []
        for i in range(starting_id,final_id):
            songs = sorted(song_service.getAllSongsForSetlistId(i),key=
    lambda x: x.pos_in_setlist,reverse=False)
            aux= plotting.get_1d_interpolation(songs, desired_attribute)
            if aux!=None:
                functions.append(aux)
        return plotting.matrix_mean_calculation(functions, first_x,
    last_x,title)
```

La final, avem 100 de valori pentru y ce le salvăm în fișier, că să nu recalculăm funcțiile la fiecare rulare. Odată ce este finalizat calculul pentru fiecare caracteristică, funcțiile arată în felul următor:

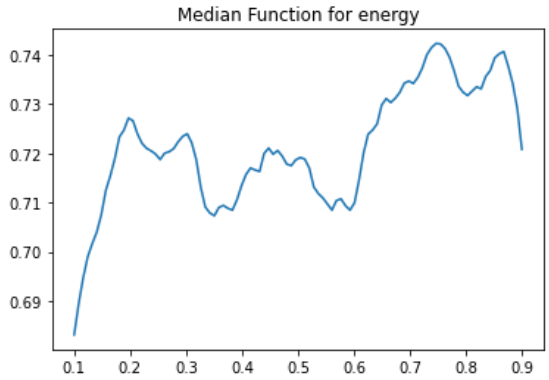

Figura 3.9: Funcția pentru ”energy”

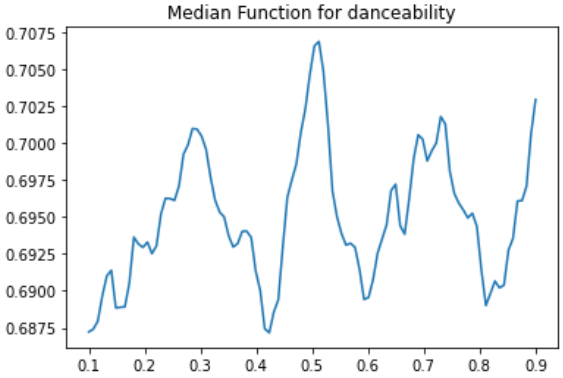

Figura 3.10: Funcția pentru ”danceability”

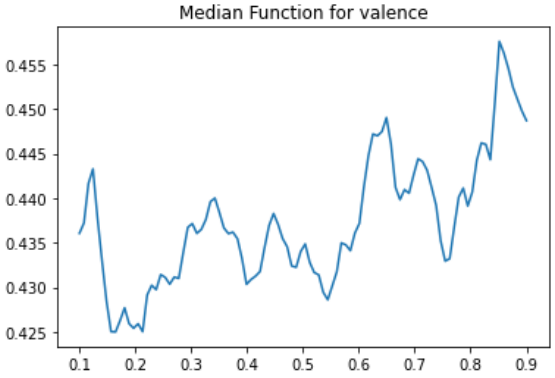

Figura 3.11: Funcția pentru ”valence”

În momentul de față, avem funcțiile caracteristicilor dorite și mai rămâne de alcătuit algoritmul genetic.

3.5 Algoritmul Genetic

Mulțumită secțiunilor din capitolul teoretic ce explică elementele de interes, implementarea algoritmului genetic devine realizabilă. Începem prin a construi o clasă "Chromosome" ce va conține următoarele atribute: representarea, fitness, tracks (domeniul de unde să aleagă piese) și perechi de valori [valoare,poziție] pentru "capetele" fiecărei caracteristici, câte un minim și un maxim pentru "energy" , "danceability" și "valence".

Pentru început, se consideră următoarele variante de operatori:

- Selecție - Alegem părinții la întâmplare, fară nicio pondere. De asemenea, cea mai bună soluție este transmisă următoarei generații.
- Încrucișare - Single-point;
- Mutație - Atribuim o șansă pentru a schimba doar doi atomi între ei și o alta șansă pentru a schimba un atom cu altul din domeniu (care este foarte posibil să nu fie prezent în nicio soluție).

Structura reprezentării cromozomului va fi o listă de dicționare, fiecare pentru câte o piesă, având toate caracteristicile prezente în fișierul "csv" de unde se aleg.

Metoda ce evaluează soluția pentru a-i acorda un fitness implementează raționamentul specificat în partea teoretică:

```
def fitness_evaluation(self,chromosome):
        energy_sum = 0
        dance_sum = 0
        valence_sum = 0

        chr_dance_diff = chromosome.get_max_dance()[0] - chromosome.
    get_min_dance()[0]
        dance_rate = chr_dance_diff / self.__dance_diff

        chr_energy_diff = chromosome.get_max_energy()[0] - chromosome.
    get_min_energy()[0]
        energy_rate = chr_energy_diff / self.__energy_diff

        chr_valence_diff = chromosome.get_max_valence()[0] - chromosome.
    get_min_valence()[0]
        valence_rate = chr_valence_diff / self.__valence_diff

        chromo = chromosome.get_repres()

        for i in range(len(chromo[:-1])):
            relative_position = 0.1 + i * (0.8 / self.__setlistSize)
```

```
19          relative_position_next =  0.1 + (i+1) * (0.8 / self.
    __setlistSize)
20
21          best_dance = dance_rate * (float(self.__danceFunction(
    relative_position_next)) - float(self.__danceFunction(
    relative_position)))
22          actual_dance = float(chromo[i+1]['danceability']) - float(
    chromo[i]['danceability'])
23          dance_score = self.evaluate_rate(actual_dance,best_dance) /
    self.__setlistSize
24          dance_sum += dance_score
25
26          best_energy = energy_rate * (float(self.__energyFunction(
    relative_position_next)) - float(self.__energyFunction(
    relative_position)))
27          actual_energy = float(chromo[i+1]['energy']) - float(chromo[i
    ]['energy'])
28          energy_score = self.evaluate_rate(actual_energy,best_energy)
    / self.__setlistSize
29          energy_sum += energy_score
30
31          best_valence = valence_rate * (float(self.__valenceFunction(
    relative_position_next)) - float(self.__valenceFunction(
    relative_position)))
32          actual_valence = float(chromo[i+1]['valence']) - float(chromo
    [i]['valence'])
33          valence_score = self.evaluate_rate(actual_valence,
    best_valence) / self.__setlistSize
34          valence_sum += valence_score
35
36      return (dance_sum+energy_sum+valence_sum)/3
37
38     '''
39     actual = actual difference between current point and the next
40     best = best difference
41
42     output : float value representing the score, best possibile score
    being 1
43     '''
44     def evaluate_rate(self,actual,best):
45         import numpy as np
46         evaluation = 0
47         if np.sign(actual)!=np.sign(best):
48             return evaluation
49         # return 0 if the actual difference has a diff sign that the best
50         # difference : it means that the trend has the opposite direction
     than
```

```
51          # the trend we want to achieve, so we give it the worst possible
    score: 0
52          evaluation = self.my_quadratic_formula(best,best-actual)
53          return evaluation
54
55
56      def my_quadratic_formula(self,denominator,x):
57          value = -(x/(denominator+0.000000001))**2 + 1
58          if value >=0:
59              return value
60          return 0
```

Ținând cont de hotărârea că ne dorim o performanță de 75%-80%, vom analiza, pe rând, diferite metode pentru operatorii din problemă.

3.5.1 Selecția

Firește, selecția menționată că fiind implementată inițial nu este o metodă foarte puternică. În continuare, nemodificand restul operatorilor, se vă testa eficienta pentru varianta intiala, o selecție turnir și o selecție prin ruleta, pentru 500 de generații, populație de 20, mărime a cromozomului (numărul de piese) de 15. Vom considera piese din genurile ”techno” și ”tech house”.

Performanța se poate urmări în următorul tabel, unde avem numele tipului de selecție, și media celui mai bun fitness pe 10 rulări:

Tip selecție	Cel mai bun fitness	Timp de rulare (sec)
Greedy	0.5396	35.02
Turnir	0.6272	34.34
Ruletă	0.5965	34.22
Ruletă și Turnir	0.6516	34.17

Tabela 3.1: Performanța obținută

Se observa ca ultima varianta ne procura cele mai bune rezultate, asa ca aceasta va ramane in algoritmul final.

3.5.2 Încrucișarea

Pentru acest operator, implementarea actuală este cea cu un punct de tăietură. Vom analiza performanța unei implementări pentru două puncte de tăietură, precum și a unei reproduceri asexuale în cazul în care descendenții rezultați sunt mai slabi decât părinții. În mod evident, implementăm varianta pentru soluții ordonate.

Testele pentru selecție au fost făcute în varianta cu un punct de tăietură, așa că scorul mediu și timpul de rulare rămân aceleași. De precizat este faptul că în cazul rar în care ar apărea o piesă de două ori, se alege aleator alta din domeniu.

Tip încrucișare	Cel mai bun fitness	Timp de rulare (sec)
1 punct	0.6516	34.17
2 puncte	0.6616	34.625
2 puncte + repr. asexuală	0.6262	34.54

Tabela 3.2: Performanța obținută

Putem remarca faptul că încrucișarea cu 2 puncte de tăietură are o performanță mai bună decât restul, așadar se rămâne la această implementare. Trebuie avut grijă, în schimb, de faptul că trebuie să trimitem mai departe descendentului care sunt minimul și maximul pentru fiecare caracteristică, astfel încât evaluarea fitness-ului să nu producă rezultate neașteptate.

3.5.3 Mutația

Având în vedere natura problemei, faptul că avem un cromozom de lungime finită mai mică decât domeniul de piese de unde se alege, putem trage concluzia că mutația ar trebui să poată face două lucruri: să schimbe atomi între ei și să introducă atomi noi în locul unora. Din nou, și aici este necesar să ținem cont de minimuri și maximuri, pentru a nu pierde rata de conversie, iar metoda de evaluare să funcționeze conform așteptărilor.

Vom acorda două șanse: una pentru a schimba doi atomi între ei, iar alta pentru a schimba atomi cu alții din domeniu. În acest caz, trebuie hotărât și câți atomi să fie schimbați, dacă acest număr să fie stabil sau dinamic, iar aici este nevoie de mai multe teste, dar putem să minimizăm numărul lor făcând următoarea deducție: Probabilitatea că o piesă întâmplătoare să se potrivească mai bine decât cea pe care o schimbă este foarte mica, din cauza faptului că aceea ce ar fi substituită a ajuns acolo pe un drum evolutiv. Așadar, șansa pentru schimbarea cu altă piesa din domeniu ar trebui să fie una destul de mică, iar în urma testelor, raționamentul pare corect. Până acum am avut cea mai bună medie de 0.66 pentru o probabilitate de 30% de a schimba 30% din populație. În urma testării cu o șansă de $\frac{100}{lungimeCromozom}$ pentru fiecare atom de a fi schimbat, avem o medie mai bună: **0.7158**.

Pentru interschimbarea a doi atomi intre ei, avem intial probabilitate de 33% de a face acest lucru. Testând cu 50%, fitness-ul mediu a devenit 0.6901, așa că se vă

rămâne pe probabilitatea inițială de 33%. Lungimea cromozomului în exemplul dat pentru a testa performanța este de 15, dar în mod normal, utilizatorul programului alege numărul de piese care să compună *setlist*-ul dorit, așa că trebuie înțeles faptul că pentru un număr mai mare, spre exemplu 50, schimbarea a doi atomi intre ei nu se simte atât de tare precum într-un *setlist* de 15. Verificând performanta pe exemplu pentru 2 interschimbari, adică 4 piese în total să își schimbe locurile, rezultă fitness mediu 0.6892. Așadar, datorită exemplului cu 15 piese și rezultatului bun, vom repeta procesul interschimbarii de n ori, unde $n = [lungimeCromozom/15]$.

3.5.4 Observații

Dat fiind faptul că timpul de rulare pentru lungime de 15 la 500 de generații este aproximativ 30 de secunde, mărim numărul final de generații la 1000. Ne putem da seama că pentru lungime mai mare, nu numai că timpul de rulare vă fi mai mare, dar și performanța vă fi mai mică. Aici intervine întrebarea că dacă și pentru un număr mic de piese avem performanță rea, iar dacă acest lucru este adevărat, care este lungimea de piese unde algoritmul alcătuit este eficient? Continuând cu exemplul de 500 de generații și populație de 20, documentăm cea mai bună soluție pentru *setlist*-uri de lungime 5 până la *setlist*-uri de lungime 40.

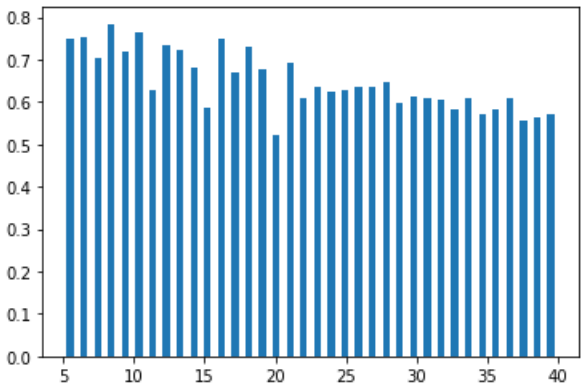

Figura 3.12: Histograma pentru cea mai buna soluție în funcție de lungimea cromozomului

Analizând histograma rezultată, se poate trage concluzia că performanța scade într-o singură direcție, cu cât cromozomul e mai lung, cu atât soluția este mai slabă, dar acest lucru nu este foarte descurajator datorită hotărârii că un scor de 75-80% este destul de bun.

3.6 Interfață Grafică

Pentru realizarea unei interfețe prin intermediul căreia utilizatorul să interacționeze

cu programul, am folosit librăria PyQT5, motivul fiind că Spyder, IDE-ul folosit, o avea inclusa. Clasa specifică interfeței face parte dintr-o arhitectura stratificata, aceasta fiind la cel mai înalt nivel. Ea comunică împreună cu un modul ce la rândul lui, apelează algoritmul genetic. În cadrul realizării bazei de date, s-au folosit și service-uri și repository-uri pentru săvârșirea cerințelor autoimpuse, acestea aflandu-se pe nivele diferite, aferente.

Interfața grafica arată în felul următor:

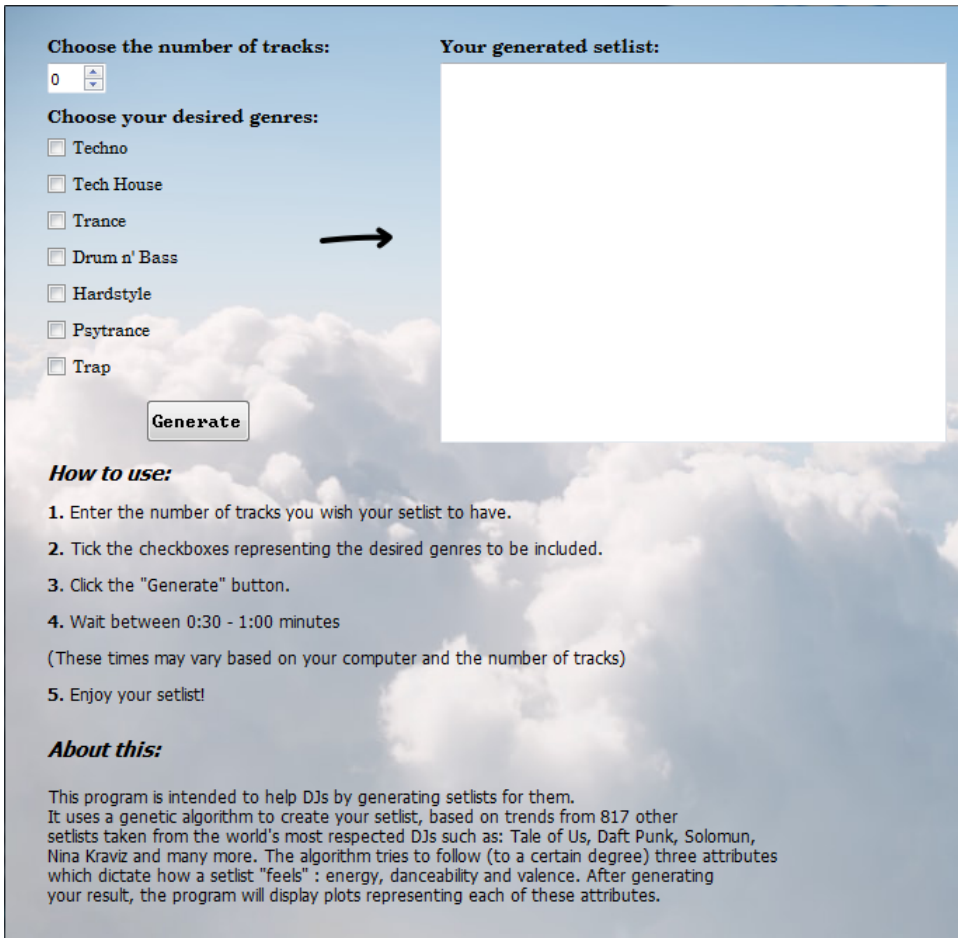

Figura 3.13: GUI-ul la pornire

Utilizatorul trebuie să introducă numărul de piese dorite, să dea click pe căsuțele de lângă fiecare gen muzical care dorește să fie prezent în *setlist*-ul sau, să apese pe butonul cu mesaj sugestiv, iar apoi să aștepte rezultatul. Acesta vă primi un mesaj de eroare informativ în cazul în care nu selectează niciun gen sau dorește un număr mai mic de 5 piese. După ce algoritmul termină rularea, în spațiul mare din dreapta îi vă apărea *setlist*-ul, numerotat în ordinea în care ar trebui prestat. De asemenea, îi vor apărea trei comparații intre graficele funcțiilor caracteristicilor luate în calcul, pentru eventuala vizualizare. Dacă utilizatorul nu este fericit cu *setlist*-ul generat,

poate apăsa din nou butonul ”Generate”.

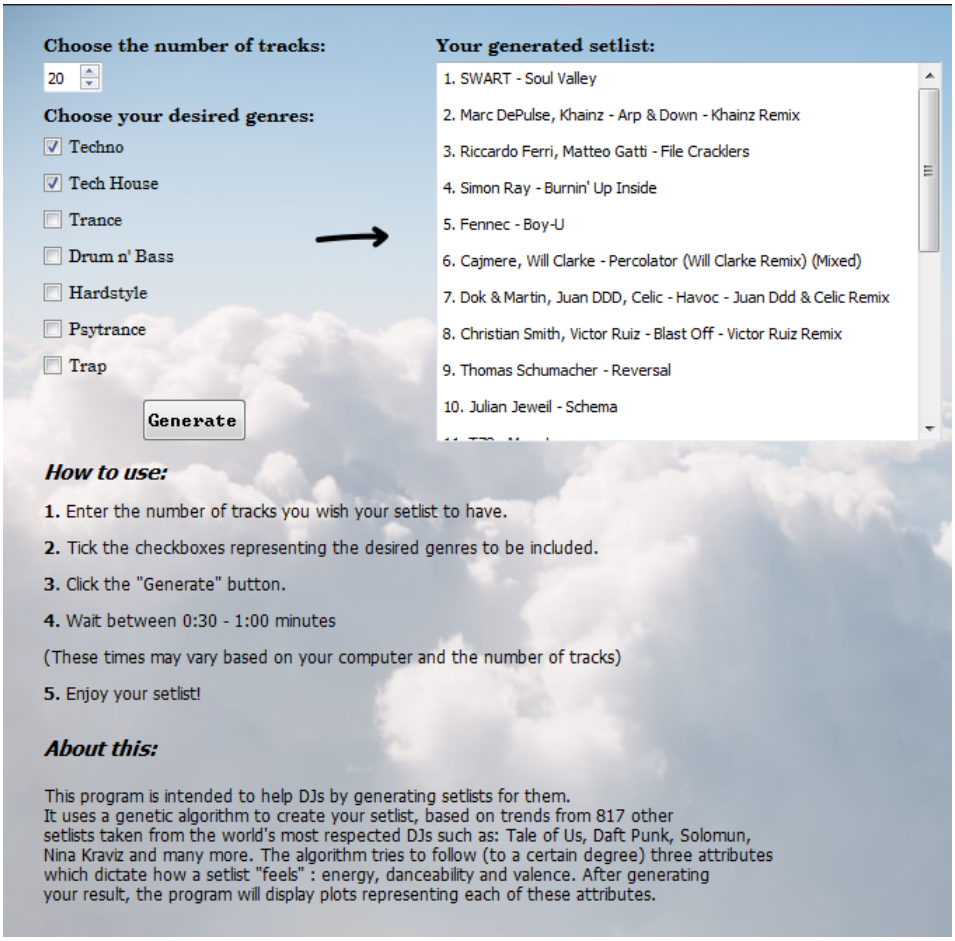

Figura 3.14: GUI-ul după utilizare

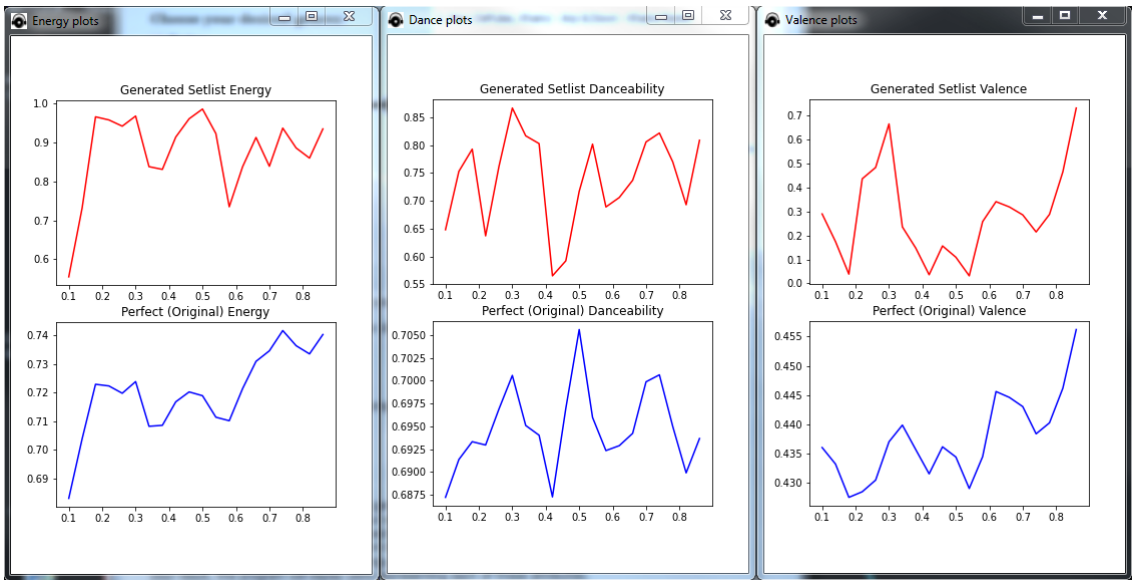

Figura 3.15: Graficele afișate după utilizare

Capitolul 4

Evaluarea Rezultatelor

4.1 Verificarea Graficelor

Programul fiind gata, mai rămâne doar evaluarea soluțiilor primite, iar prima modalitate prin care putem face acest lucru este simpla comparare a graficelor. După o rulare de un minut din interfața grafică pentru un *setlist* de 15 piese cu genurile ”techno” și ”tech house”, programul ne-a oferit următorul *setlist*:
1. Shlomi Aber - Forum
2. Crystal Waters, RobbieG - Gypsy Woman (She’s Homeless) - RobbieG Remix / Radio Edit
3. Peter Brown - Say It Again
4. Chris Veron - Panic Lock - Original Mix
5. Vangelis Kostoxenakis - Been Long Time
6. Keith Carnal - Instantaneously (Subjected Rave Edit)
7. Adam Beyer - That Would Be the Sun
8. Ernest & Frank - Superstition
9. Christian Cambas, T78 - The Outsiders - T78 Remix
10. Alderaan - Ancient Particles - Original Mix
11. Mendo - Get A Funk
12. Șlam - Viper
13. Ryan Mckay - Cipher
14. Anetha - Acid Train
15. Lowkey, Kardinal - Flashover

Graficele *setlist*-ului arată în felul următor:

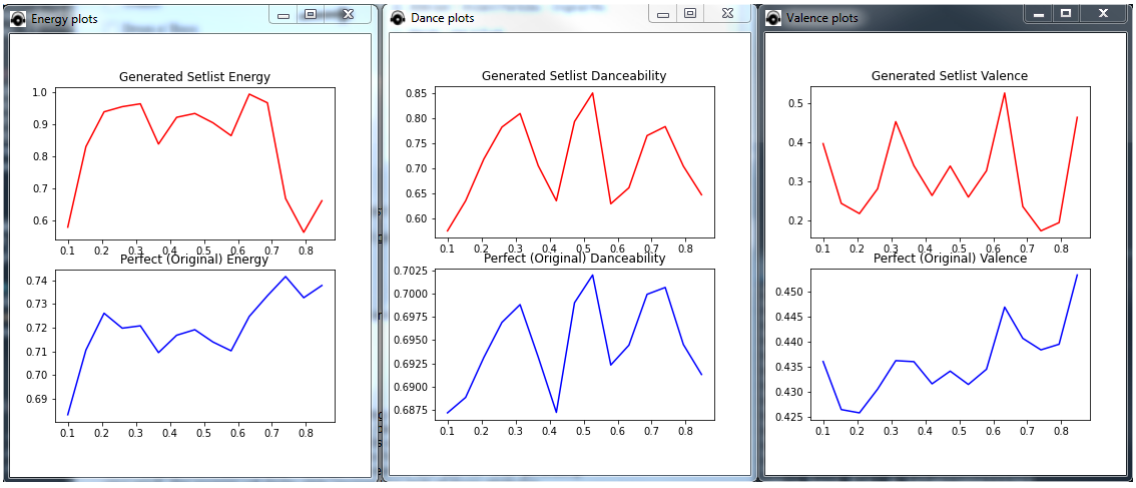

Figura 4.1: Energy, danceability, valence pentru *setlist*-ul rezultat

Se pot observa similarități clare între graficele *setlist*-ului rezultat și cele pe care algoritmul a dorit să le imite, adică graficele funcțiilor calculate pentru 810 *setlist*-uri deja făcute de *DJ*. Din acest punct de vedere, algoritmul reușește să facă ceea ce s-a propus în problemă. De exemplu, pentru "danceability", se văd cele trei creșteri distincte, pentru "energy" se distinge "podul" din mijloc, iar pentru "valence" arată în totalitate aproape la fel, atât că ridicăturile sunt mai accentuate.

Singura problemă este că nu putem valida algoritmul doar prin comparația unor grafice, *setlist*-ul trebuie testat cu adevărat.

4.2 Testarea Setlist-ului

Dat fiind faptul că în problemă am specificat că ne dorim crearea unui algoritm ce poate genera *setlist*-uri în așa manieră precum ar face-o un *DJ*, rezultatul trebuie ascultat că în contextul unui spectacol. După crearea unui playlist pe Spotify cu piesele date în ordinea lor, am trimis link-ul mai multor persoane și după feedback-ul primit, am constatat două lucruri: primul fiind că într-adevăr, playlist-ul iți da acea senzație, că și cum ai fi la un performance de *DJ*, iar al doilea este că unii și-ar fi dorit să audă și niste piese mai cunoscute. Într-adevăr, algoritmul nu discriminează în funcție de popularitatea pieselor, deoarece nu am dorit posibila apariție repetată a unor piese în soluțiile generate. Totuși, prima constatare confirmă validitatea algoritmului.

Capitolul 5

Concluzii

5.1 Posibilități de Extindere

Acest program, cu toate că și-a îndeplinit scopul propus, poate fi extins în continuare. Unele lucruri ce pot fi făcute pe viitor sunt:

- Implementarea unei probabilități de selecție a pieselor din domeniu în funcție de popularitatea lor;
- Adăugarea mai multor piese în domeniu din genurile existente sau din alte genuri muzicale;
- Aplicație web, pentru folosirea algoritmului fară a descărca nimic;

5.2 Deznodământ

Această lucrare a pornit de la ipoteza că setlist-urile DJ-lor se ”simt” într-un anumit fel și a îndeplinit scopul de a genera setlist-uri conform tendințelor DJ-lor. Voiajul dinspre abstract și teoretic către numere și practică a fost unul ce a necesitat planificare și raționament, pentru a nu porni pe o cărare greșită, ci a reuși a duce la bun final proiectul. Că și notă personală, pot spune că un mare câștig din realizarea problemei a fost dobândirea abilitații de a împarți și organiza sarcinile în altele mai mici, de a nu renunță în cazul apariției unui obstacol și de a găsi un motiv și semnificație la orice pas spre final.

Bibliografie

[1] 2021 Spotify AB. *Spotify Web API Reference*. https://developer.spotify.com/documentation/web-api/reference/. Online; accesat la data de 18 Aprilie 2021.

[2] NumPy Community. *NumPy Reference*. Release 1.20.0. Ianuarie 2021.

[3] Laura Dioșan. *Curs Inteligență Artificială*. Universitatea Babeș-Bolyai, Facultatea de Matematică și Informatică. **april** 2020.

[4] Edwin Herman Gilbert Strang. *Areas between Curves*. https://math.libretexts.org/@go/page/2519. Online; accesat la data de 26 Aprilie 2021.

[5] Charles R. Harris, K. Jarrod Millman **and** Stéfan J. van der Walt. "Array programming with NumPy". **in**: *Nature* 585.7825 (**september** 2020), **pages** 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[6] DKA Jong. *Are Genetic Algorithms Function Optimizers?. Parallel Problem Solving from Nature 2, Manner, R. and Manderick, B. eds*. 1992.

[7] Paul Lamere. *Spotipy Documentation*. https://spotipy.readthedocs.io/en/2.18.0/. Online; accesat la data de 20 Aprilie 2021.

[8] Melanie Mitchell. *An Introduction to Genetic Algorithms*. 2nd. USA: MIT Press, 1998. ISBN: 0262631857.

[9] Leonard Richardson. *Beautiful Soup Documentation*. https://spotipy.readthedocs.io/en/2.18.0/. Online; accesat la data de 15 Aprilie 2021.