

ISS projekt 2025/26
Protokol k projektu
“Music search”

Andrej Bližnák - xblizna00

1. Úvod a analýza problému.....	2
1.1. Cieľ.....	2
1.2. Princíp.....	2
1.3. Dáta.....	2
2. Spracovanie signálu.....	3
2.1. Rámcovanie.....	3
2.2. Váhovanie.....	3
2.3. Spektrum.....	3
2.4. Magnitúdová zložka a filtrácia.....	4
2.5. Logaritmické spektrum.....	4
3. Finger-printing.....	5
3.1. Hľadanie vrcholov.....	5
3.2. Hašovanie.....	6
4. Vyhľadávanie a optimalizácia.....	7
4.1. Inverzný index.....	7
4.2. Optimalizácia databázy.....	7
5. Skórovanie a Časové zarovnanie.....	7
5.1. Histogram Offsetov.....	7
6. Výsledky.....	8
6.1. Google Colab.....	8
6.2. Python (Lokálne).....	8
7. Predchádzajúce pokusy.....	9
7.1. Nultý pokus - Výpočet energie.....	9
7.1.1. Metóda.....	9
7.1.2. Výsledok.....	9
7.1.3. Zhodnotenie.....	9
7.2. Prvý pokus - Priemerovanie spektrogramu.....	9
7.2.1. Metóda.....	9
7.2.2. Výsledok.....	9
7.2.3. Zhodnotenie.....	10
7.3. Druhý pokus - Mel-spektrogram.....	10
7.3.1. Metóda.....	10
7.3.2. Výsledok.....	10
7.3.3. Zhodnotenie.....	10
7.4. Tretí pokus - Audio Fingerprinting a Jaccard.....	10
7.4.1. Metóda.....	10
7.4.2. Výsledok.....	10
7.4.3. Zhodnotenie.....	11
7.5. Finálne riešenie - Histogram Offsetov.....	11
7.5.1. Vylepšenie.....	11
7.5.2. Dôvod.....	11
8. Záver.....	11
9. Reference.....	13

1. Úvod a analýza problému

1.1. Cieľ

Vytvoriť systém na vyhľadávanie hudby na základne krátkych nahrávok.

1.2. Princíp

Vypracovanie je založené na audio fingerprinting (otlačkoch zvuku), inšpirované algoritmom Shazam s drobnými úpravami pre zjednodušenie.

Hlavnou myšlienkou nie je porovnávať zvukové vlny priamo, pretože tie sú citlivé na šum a skreslenie, ale extrahovať z nahrávky nejaké unikátne charakteristiky.

Celý algoritmus funguje v štyroch krokoch:

1. Časovo-frekvenčná analýza

Signál sa prevedie do frekvenčnej domény, aby sme videli, aké tóny hrajú v akom čase.

2. Konštelačná mapa

V spektrogramme sa vyhľadávajú iba tie najvýraznejšie body (lokálne maximá).

3. Kombinatorické hašovanie

Samotné maximum nie je unikátne. Unikátny je až vzťah medzi dvoma maximami. Takzvaný "kotviaci bod" a "susedný bod" vytvárajú odtlačky (haše). Haš v sebe kóduje frekvencie oboch bodov a časový rozdiel medzi nimi.

4. Časové zarovnanie

Pri vyhľadávaní nestačí nájsť zhodné haše. Musí sa overiť, či nasledujú v správnom časovom poradí. To sa robí pomocou histogramu časových posunov. Ak ide o tú istú nahrávku, všetky zhodné haše budú mať konštantný časový posun voči originálu v databáze.

1.3. Dáta

Vstupom sú `.wav` súbory, ktoré spracovávame vo frekvenčnej doméne.

2. Spracovanie signálu

2.1. Rámcovanie

Slúži na rozdelenie signálu na krátke úseky pre potreby časovo-frekvenčnej analýzy.

Nastavujeme pri tom dĺžku `Window size` jedného okna, ktorá určuje rozhranie medzi frekvenčným a časovým rozlíšením a prekrytie `Overlap`, ktoré zabraňuje stratám informácií na okrajoch rámcov.

Ja som signál rozdelil na časové úseky, konkrétnie používam 64 milisekundovú dĺžku (1024 vzorkov pri frekvencii 16kHz) pre jedno okno s prekrytím (overlap) 50 percent, čo zodpovedá posunu o 32 milisekúnd. Tieto hodnoty mi prišli vhodné aj výkonnostne ako aj presné, dosahujú 100% úspešnosť.

`Prečo ?...`

Základná Fourierova transformácia predpokladá, že frekvenčný obsah signálu sa v čase nemení. Hudba je dynamická, jej frekvencie sa neustále striedajú, preto ju musíme rozkúskovať na milisekundové úseky `Rámce`.

Dĺžku okna som počas projektu viackrát zmenil. Platí totiž pravidlo, že čím je okno kratšie, tým presnejšie vieme určiť čas, ale o to horšie sa určuje frekvencia, takže trebalo to vybalancovať. Nenašiel som však jednotné informácie, ktoré by určovali konkrétnu "veľkosť" okna používanú v Shazame, avšak menšie veľkosti s ktorými som pôvodne pracoval (20ms/32ms) mi nedali presné výsledky a sú skôr vhodné pri práci s rozpoznávaním reči, teda pri rýchlejších zmenách signálu.

Rovnako tak som zvolil pôvodne 75% `Prekrytie`, avšak bolo to výkonnostne náročné, generovalo to dvakrát toľko rámcov a pre riešenie tohto projektu mi stačilo používať len 50% aby som dosiahol 100% presnosť.

2.2. Váhovanie

Tu používam Hanningove okno, priamo funkciu `np.hanning`, aby som potlačil spektrálny únik, ktorý vzniká na okrajoch rámcov. Zjemňuje prechody na krajoch okna do nuly.

2.3. Spektrum

Použil som Fourierova transformáciu čisto s reálnymi číslami, na vstup posielam 2048 vzoriek.

Prečo?...

Audio signál obsahuje iba reálne čísla, obyčajná FFT by produkovala symetrický výsledok, kde druhá polovica by bola zrkadlovým obrazom tej prvej. Práve použitím RFFT tomu vieme predísť, pretože tento algoritmus je optimalizovaný priamo na reálne vstupy a nepočítá zbytočnú zrkadlovú (symetrickú) časť, čím zrýchlime výpočet.

Aj keď jeden rámec má dĺžku 1024 vzorkov, používam dvakrát toľko pri vstupe do FFT. Táto technika sa nazýva **Zero Padding** (doplnenie nulami). Týmto dosiahнем jemnejšie delenie frekvenčnej osi, čo sa odzrkadluje neskôr pri hľadaní "maxím" vo funkcií **find_peaks**.

2.4. Magnitúdová zložka a filtrácia

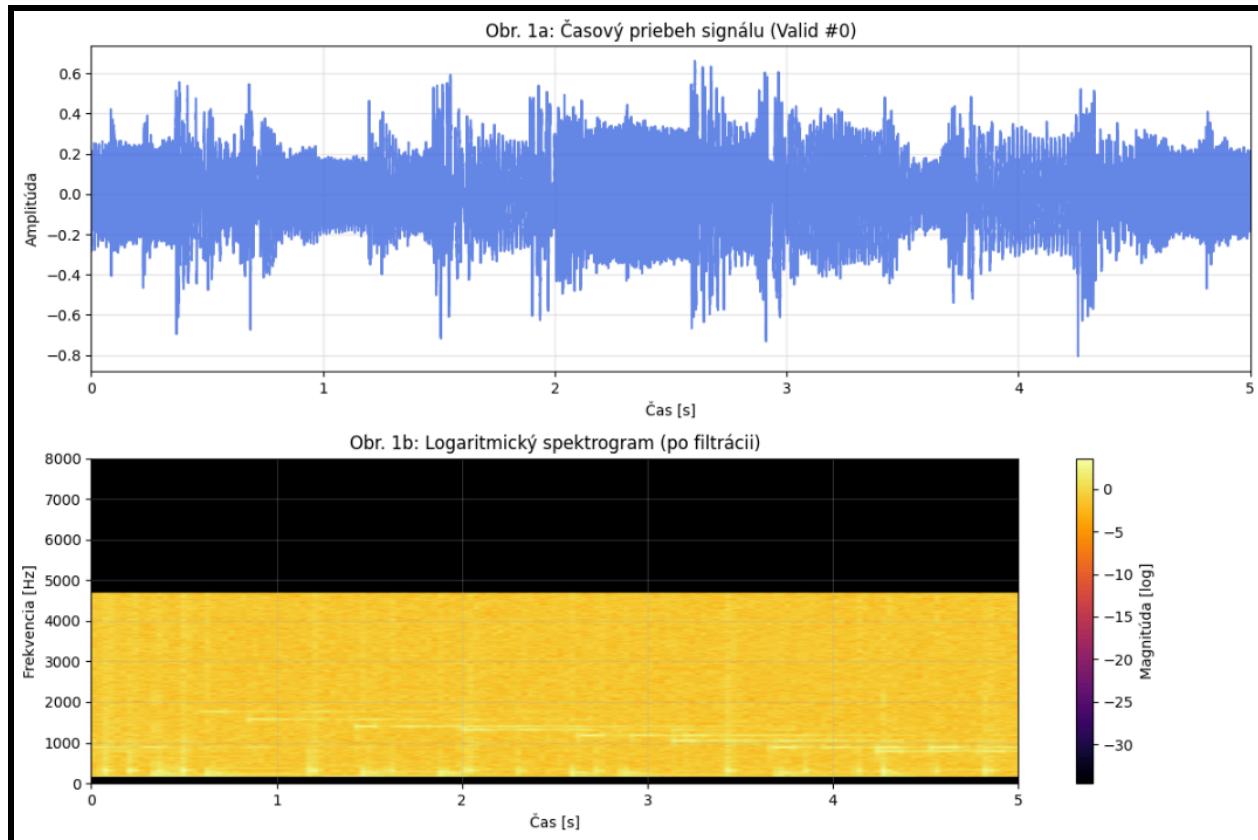
Výstupom z RFFT sú komplexné čísla. Pre vytvorenie spektrogramu nás však nezaujímajú fázové posuny. Preto som vypočítal **magnitúdu** (absolútne hodnotu) každého prvku. Rovnako tak nás nezaujímajú ani príliš nízke alebo vysoké frekvencie (väčšinou ide len o šum), takže opäť pre optimalizáciu sa ich môžeme v poriadku zbaviť.

2.5. Logaritmické spektrum

Vypočítanú magnitúdu spektra som previedol na logaritmickú škálu.

Prečo?...

Obyčajné hodnoty v spektre majú veľký rozsah. Silné tóny majú svoje hodnotu naozaj vysoko, zatiaľ čo tichšie sú blízko nuly. Logaritmus tento rozsah stlačí, čím sa zvýraznia spektrálne špičky. Toto nám opäť pomôže pri hľadaní "maxím".



Obrázok 1: Ukážka spracovania signálu. Hore je časový priebeh 5 sekundovej testovacej nahrávky. Dole je jej výsledný logaritmický spektrogram po aplikácii RFFT a filtričii. Jasnejšie farby indikujú vyššiu energiu na danej frekvencii.

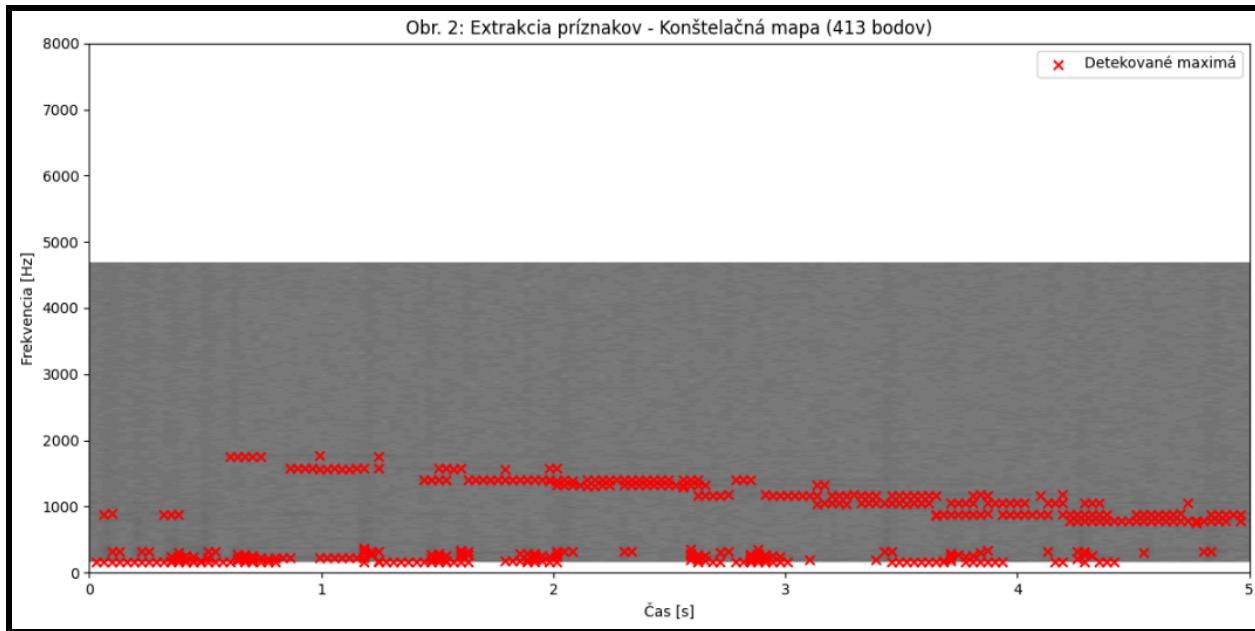
3. Finger-printing

3.1. Hľadanie vrcholov

Na nájdenie lokálnych extrémov využívam knižnicu `scipy.signal`. Algoritmus prechádza spektrogram po `rāmcoch`. Pre každý nastavím `prah` (threshold) na 50% maximálnej hodnoty.

Prečo?...

Pôvodne som zvažoval statický/fixný prah, ale hlasitosť nahrávok sa mení. Relatívny prah ($0.5 * \text{max}$) sa dynamicky prispôsobuje hlasitosti konkrétnego momentu v piesni. Týmto spôsobom filtrujem šum pozadia a vyberám len dominantné frekvencie, ktoré tvoria melódiu. Opäť, skúšal som rôzne hodnoty. Čím nižšie som išiel, dostával som príliš veľké množstvo bodov, čo by mohlo viesť k vyššej presnosti ale výrazne to spomalovalo výpočet. Zase naopak vyššie hodnoty viedli k menšiemu množstvu detekovaných bodov.



Obrázok 2: Konštelačná mapa. Na pozadí je spektrogram z prvého obrázku. Červené krížiky označujú maximá (peaky), ktoré prekonali dynamický prah. Tieto body slúžia ako základ pre tvorbu hashov.

3.2. Hašovanie

Tu vytváram odtlačky, čím zabezpečujem identifikáciu. Z poľa najdených maxím vytváram skupinky frekvencii a časového rozdielu. Iterujem cez všetky body a každý bod identifikujem ako **Anchor point**. Priradím jej 5 za ňou nasledujúcich bodov **Neighbors**. Z každej dvojice

(Anchor -> Neighbor) vytvorím **hash**, ktorý sa skladá z troch častí:
 $\text{hash} = (\text{frekvencia(Anchor}), \text{frekvencia(Neighbor}), \text{čas(Delta)})$,
pričom $\text{Delta} = \text{čas(Neighbor)} - \text{čas(Anchor)}$. Výsledok ešte orezávame, aby sa zmestil do 32 bitov.

Prečo?...

Najdôležitejším prvkom je tu časový rozdiel **Delta**. Ak by sme do **hash**-u vložili presný čas, systém by nefungoval. Pomocou tohto časového rozdielu sme dosiahli, že vzťah medzi dvoma tónmi zostane rovnaký, aj keď nahrávku začнем nahrávať v strede pesničky. Počet susedov som opäť volil experimentálne, pri menšom počte bolo málo dát, teda horšie výsledky, pri väčšom počte som výsledky ani nedostal kvôli vysokej náročnosti.

4. Vyhľadávanie a optimalizácia

4.1. Inverzný index

Po vygenerovaní fingerprintov pre všetkých 706 známych a 50 validných skladieb sme vytvorili vyhľadávaciu **databázu** (inverzný index). Ide o štruktúru, kde kľúčom je **hash** a hodnotou je zoznam všetkých výskytov tohto hashu v tvare: `hash_dt = (song(ID), shown_time)`.

Prečo?...

Ak by sme nepoužili túto štruktúru, museli by sme pre každý **hash** z testovacej nahrávky iterovať cez všetky **known** signály, čo opäť spomaluje výpočet. Vďaka **databáze** viem so zložitosťou $O(1)$ pristúpiť k hľadanému úseku.

4.2. Optimalizácia databázy

Toto bola pre mňa kritická zmena, ktorá mi zrýchliala beh programu z niekoľkých minút na niekoľko sekúnd. Rozhodol som sa odstrániť **hashe**, ktoré sa vyskytovali príliš veľakrát, konkrétnie 1% najčastejších.

Prečo?...

V hudbe existuje niečo ako **hashe**, reprezentujúce ticho, biely šum. Nachádzajú sa takmer v každom signáli veľmi veľakrát. Tým že nie sú unikátne, neprispevajú na identifikáciu skladieb a preto sa ich môžeme zbaviť. Opäť, dalo by sa povedať, že v určitých signáloch kde sa nevyskytujú by boli možno kľúčové pre presnosť, no z hľadiska náročnosti spôsobovali väčšinu spomaľovania a preto som sa rozhodol sa ich zbaviť s rizikom zníženia presnosti (k zníženiu presnosti pri testovaných súboroch nedošlo).

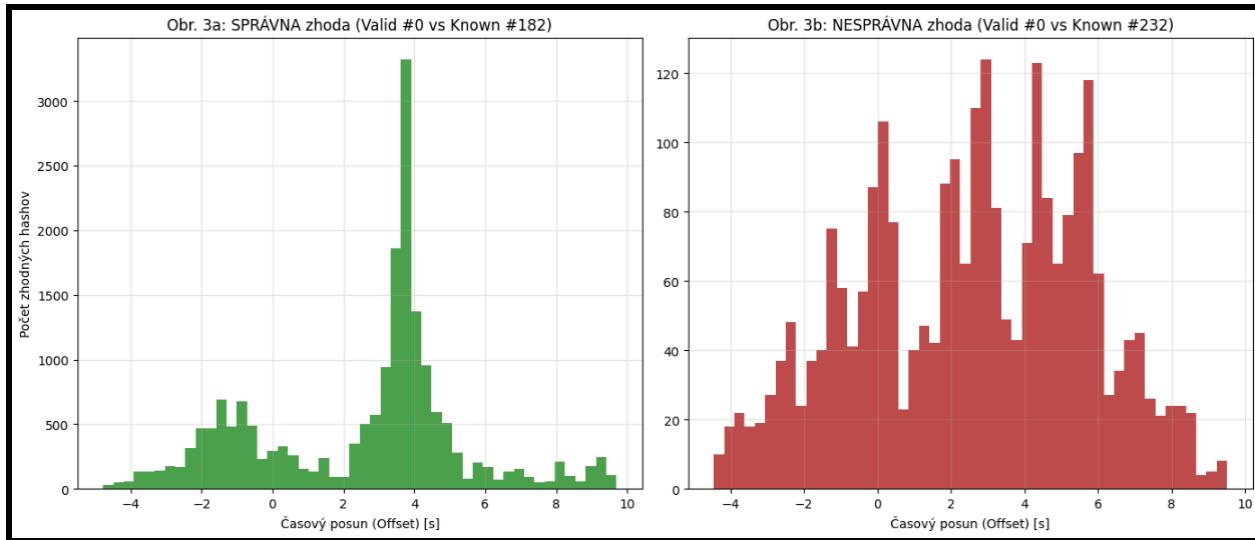
5. Skórovanie a Časové zarovnanie

5.1. Histogram Offsetov

Ked' nájdem zhodu **hashu** medzi testovacou nahrávkou a známou skladbou, nepočítam len obyčajný súčet zhôd. Počítam **časový posun** (Offset). `Offset = time(known) - time(valid)`. Výsledné skóre pre danú nahrávku je počet zhôd, ktoré majú rovnaký časový **offset**.

Prečo?...

Je to niečo ako finálny filter proti náhodným zhodám. Môže sa stať, že dva **hashe** sa náhodou zhodujú, ale sú v inom čase. Ak však ide o tú istú nahrávku, všetky zhodné hashe musia byť posunuté o rovnakú konštantu. Ignorovaním náhodných zhôd s rôznymi posunmi som dosiahol o niečo väčšiu presnosť.



Obrázok 3: Vľavo je histogram offsetov pri porovnaní testovacej nahrávky so správnou skladbou v databáze. Je vidieť výrazný vrchol, ktorý dokazuje, že väčšina hashov je posunutá o konštantný čas. Vpravo je porovnanie s nesprávnou skladbou, kde sú zhody hashov náhodné a netvoria dominantnú top.

6. Výsledky

6.1. Google Colab

Top 1 accuracy 100.0 %, Top 5 accuracy 100.0 %

Spustenie	Čas behu programu
1.	47 sekúnd
2.	56 sekúnd
3.	53 sekúnd

6.2. Python (Lokálne)

Neskôr som svoje vypracovanie rozbehol lokálne v pythone, rýchlosť sa zvýšila, presnosť ostala rovnaká.

(OS: Windows 11)
 (CPU/RAM: Intel(R) Core(TM)
 i5-13600K, 32GB RAM)

Top 1 Accuracy (Valid): 100.00 %
Top 5 Accuracy (Valid): 100.00 %
Celkový čas behu: 37.26 s

7. Predchádzajúce pokusy

7.1. Nultý pokus - Výpočet energie

7.1.1. Metóda

Ako prvý krok som skúsil porovnať signály bez akejkoľvek transformácie do frekvenčnej domény. Pre každú nahrávku som

$$E = \sum_n |x[n]|$$

vypočítal jediné číslo, sumu absolútnych hodnôt amplitúd vzoriek. Následne som hľadal nahrávku s najpodobnejšou hodnotou "E".

7.1.2. Výsledok

Presnosť	%
Top-1	0.1
Top-5	3

7.1.3. Zhodnotenie

Tento pokus bol maximálne neúspešný, no poslúžil na potvrdenie predpokladu, že surové časové štatistiky sú pre MIR nepoužiteľné.

7.2. Prvý pokus - Priemerovanie spektrogramu

7.2.1. Metóda

Signál som rozdelil na rámce, vypočítal FFT a získal magnitúdové spektrum.

Následne som spravil priemer cez časovú os `np.mean(axis=0)`, čím som pre každú skladbu získal jeden dlhý vektor obsahujúci "priemerné frekvenčné zloženie". Podobnosť bola vypočítaná pomocou kosínusovej podobnosti (Cosine similarity).

7.2.2. Výsledok

Presnosť	%
Top-1	4
Top-5	8

7.2.3. Zhodnotenie

Tento prístup zlyhal, priemerovaním sme stratili informácie o čase.

7.3. Druhý pokus - Mel-spektrogram

7.3.1. Metóda

Aby som sa priblížil k ľudskému vnímaniu zvuku, použil som Mel-filterbank (banka obsahujúca 40 filtrov) na transformáciu spektra do Melovej škály a aplikoval logaritmus pre zrazenie vysokých/nízkych hodnôt. Potom nasledovalo opäť priemerovanie cez čas.

7.3.2. Výsledok

Presnosť	%
Top-1	44
Top-5	54

7.3.3. Zhodnotenie

Výrazné zlepšenie oproti prvému pokusu potvrdilo, že Melova škála a logaritmovanie spektra lepšie reprezentujú audio signál. Metóda však stále ignorovala časovú následnosť nahrávok.

7.4. Tretí pokus - Audio Fingerprinting a Jaccard

7.4.1. Metóda

Prešiel som na detekciu lokálnych maxím (peaks) a tvorbu hashov z trojíc frekvencia(anchor), frekvencia(neighbor), time(delta), podobne ako vo finálnej verzii. Rozdiel bol v porovnávaní. Namiesto kontroly časového posunu som používal Jaccardov index (priek množín hashov delený ich zjednotením).

7.4.2. Výsledok

Presnosť	%
Top-1	98
Top-5	100

7.4.3. Zhodnotenie

Tento skok v presnosti potvrdil, že kľúčom k úspechu je kombinatorické hašovanie, ktoré zachytáva lokálnu štruktúru (vzťah medzi susednými tónmi). Táto verzia sa da považovať za úspešnú pre riešenie tohto projektu, ale stále bolo čo zlepšovať.

7.5. Finálne riešenie - Histogram Offsetov

7.5.1. Vylepšenie

Oproti tretiemu pokusu som nahradil Jaccardovu podobnosť Histogramom časových posunov, ten mi vrátil 100% presnosť a postupne som už len "rýchlosťne" optimalizoval kód.

7.5.2. Dôvod

Jaccardov index iba počíta, či sa hashe vyskytujú v oboch nahrávkach, ale nekontroloval, či sú v správnom poradí. Histogram Offsetov odstránil aj posledné falošné zhody, čím som dosiahol o niečo vyššiu presnosť.

8. Záver

Riešiť tento projekt bolo pre mňa veľmi zábavné, hlavne si myslím že som získal nové vedomosti počas riešenia rôznych situácií s ktorými som sa v projekte stretol. Som rád, že ako jeden z mála projektov na FITe má priamo v zadaní napísané "**Cílem projektu není 100% úspěšnost, ale pochopení a popis toho, co deláte.**"

Počas práce na projekte som postupoval intuitívne, zo začiatku som čerpal z vlastných vedomostí, neskôr som bol nútený si prečítať ďalšie materiály ako prezentácie z predmetu ISS, či články na internete rozoberajúce spracovanie zvuku. Zo začiatku som sa snažil písat všetky algoritmy/výpočty manuálne, ale časom sa ukázalo, že predstavané funkcie urýchlia môj program a nikde neboli zakázané. Príkladom môže byť napríklad algoritmus na hľadanie lokálnych maxím (Obrázok 4).

```

# 4. # Hľadanie lokalnych maxim
def search_peaks(spectrogram):
    peaks = []

    for time in range(spectrogram.shape[0]):
        max = find_maximum(spectrogram[time])
        for frequency in max:
            peaks.append((frequency, time))
    return peaks

def find_maximum(frame):
    prev_val = frame[ : -2]
    curr_val = frame[1 : -1]
    next_val = frame[2 : ]

    local_max = (curr_val > prev_val) & (curr_val > next_val)
    peaks_peaks_peaks = np.where(local_max)[0] + 1 # Je to tuple , treba zeropadd
    return peaks_peaks_peaks

```

Obrázok 4: Manuálne hľadanie lokálnych maxím, neskôr nahradené funkciou `find_peaks` z knižnice `scipy`.

Veľkou výzvou a najväčším “konzumentom” času bolo práve optimalizovanie.

Pôvodné implementácie ako napríklad (Obrázok 4) boli funkčné, no pomalé.

Implementáciou odstránenia najvyššieho percenta vyskytovaných hashov sa mi podarilo zbaviť sa značného množstva nepotrebných hashov pre výsledok a tým urýchliť výsledné porovnávanie.

Projekt hodnotím pozitívne. Myslím si, že sa mi podarilo vytvoriť ako-tak funkčný program, ktorý pre valid súbory dosahuje 100% úspešnosť, ale zároveň som si prakticky vyskúšal teóriu získanú z predmetu ISS. Kód som odovzdal v dvoch variantách, lokálnu verziu v pythone so všetkým potrebným pre spustenie, ako aj colab verziu. Viac informácií ohľadom odovzdaneho folderu nájdete v README.txt.

9. Reference

1. WANG, Avery Li-Chun. An Industrial-Strength Audio Search Algorithm. In: *ISMIR*. 2003. s. 7-13.
Zdroj: <https://www.ee.columbia.edu/~dpwe/papers/Wang03-shazam.pdf>
2. Will Drevo - Audio Fingerprinting with Python and Numpy
Zdroj: <https://willdrevo.com/fingerprinting-and-audio-recognition-with-python/>
3. Documentation - scipy.signal.find_peaks a scipy.fft.rfft
Zdroj: <https://numpy.org/doc/stable/reference/generated/numpy.hanning.html>
4. Roy van Rijn - Creating Shazam in Java
Zdroj: <https://www.robynrijn.com/blog/2010/06/creating-shazam-in-java/>