SONIQ

Bas van den Boom

Ruisbestendige herkenning van digitale audiofragmenten

Profielwerkstuk Commanderij College VWO 2018

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Vak: Natuurkunde  Begeleider: JGW Voets |  | https://soniq.z3r0byteapps.eu  info@z3r0byteapps.eu |
|  |  |  |

Inhoud

[Probleemdefinitie 1](#_Toc532121677)

[Programma van Eisen 2](#_Toc532121678)

[Theorie 4](#_Toc532121679)

[Plan van aanpak 9](#_Toc532121680)

[Ontwerpproces 13](#_Toc532121681)

[Testen 31](#_Toc532121682)

[Conclusie 38](#_Toc532121683)

[Evaluatie 40](#_Toc532121684)

[Nawoord 41](#_Toc532121685)

[Verwijzingen 42](#_Toc532121686)

[Bijlagen 44](#_Toc532121687)

Probleemdefinitie

Het probleem

Stelt u zich de volgende situatie voor: u bent op een feestje en de dj zet een nummer op wat u erg bevalt, maar u weet niet hoe het nummer heet. U kunt dan naar de dj toegaan om te vragen hoe het lied heet, maar dan moet u zich eerst door een grote menigte mensen heen wringen. Het zou veel makkelijker zijn wanneer u met uw smartphone in een paar tellen kunt vinden hoe het nummer heet. In dit verslag wordt beschreven hoe een dergelijk systeem werkt en hoe dit gebouwd kan worden.

Functieomschrijving

De gemaakte software moet in staat zijn kleine delen van een groter audiofragment te kunnen herleiden naar het origineel. Dit moet zelfs mogelijk zijn met achtergrondgeluid en eventuele compressie, die hoogstwaarschijnlijk aanwezig zal zijn wanneer het audiofragment met een smartphone wordt opgenomen. Verder moet het in staat zijn grote hoeveelheden originelen snel te doorzoeken, aangezien de gebruiker niet lang wil wachten.

Programma van Eisen

De gebouwde software moet aan een aantal eisen voldoen. Ten eerste moet het in staat zijn om uit kleine fragmenten van maximaal 10 seconden het origineel te kunnen herkennen. Er is voor een tijdslimiet van 10 seconden gekozen, omdat de gebruiker niet te lang wil wachten. Verder zorgt deze relatief korte tijdslimiet ervoor dat de kans op overbelasting kleiner wordt, aangezien het hierdoor niet mogelijk is een oneindige stroom van data te sturen die de software bezig kan houden en zo zorgt voor langere wachttijden voor andere gebruikers.

Ten tweede is het belangrijk dat de software vanaf een smartphone te gebruiken is. Denk hierbij aan een app die de gebruiker kan downloaden en waarmee hij of zij vervolgens een geluidsopname kan laten analyseren.

Ten derde moet de catalogus van de software groot zijn, zodat de kans dat een opgevraagd nummer herkend kan worden groot is. De software moet hier dan ook mee om kunnen gaan en moet wenselijke resultaten kunnen produceren, zelfs met grote hoeveelheden data. Er is gekozen voor een minimale capaciteit van 2000 liedjes, omdat dit een redelijk grote hoeveelheid is en dit technisch ook haalbaar moet zijn. Verder is dit getal makkelijk, omdat dan bijvoorbeeld de Top 2000 gedownload kan worden voor in de catalogus en men dan met één download voldoende nummers heeft.

Een vierde punt is de verwerkingstijd. Het is belangrijk dat de gebruiker niet lang moet wachten op het resultaat van zijn of haar zoekopdracht. Er is gekozen voor een gemiddelde verwerkingstijd van maximaal twee seconden, waarvan minimaal 90% maximaal drie seconden mag zijn. Uit onderzoek (Akamai, 2015) is namelijk gebleken dat bijna de helft van de gebruikers een laadtijd van onder de twee seconden verwacht. Hiermee wordt alleen de tijd bedoeld van het zoeken zelf, dus van opname tot resultaat, de opname zelf en het uploaden hiervan tellen niet mee omdat netwerksnelheden erg variabel zijn.

Ten vijfde is het belangrijk dat ook fragmenten die achtergrondgeluid of ruis bevatten ook herkend kunnen worden. Een belangrijke factor hierin is de signal-to-noise ratio, oftewel de signaal-ruisverhouding (Wikipedia, 2018b), wat aangeeft wat de verhouding is tussen het gewenste signaal en de rest (het achtergrondgeluid en ruis). Bij een ratio van 0dB moet de software in minimaal 80% van de gevallen het origineel kunnen herkennen. Bij waardes vanaf 3 dB moet er in minimaal 95% van de gevallen het origineel herkend worden. Bij waardes van -3dB moet nog in minimaal 70% van de gevallen het origineel herkend kunnen worden. Bij waardes van minder dan -3dB zijn er geen eisen meer voor de herkenning.

Als laatste is het nog belangrijk dat de software zo min mogelijk fout-positieven genereert. Er is een limiet gesteld van maximaal 5% aan fout-positieven, omdat dit geacht wordt technisch haalbaar te zijn voor dit project. Een fout-positief treedt op wanneer het programma aangeeft een match te hebben gevonden terwijl het te matchen nummer niet in de database staat.

Lijst van eisen:

* De geluidsfragmenten die gematcht moeten worden mogen maximaal 10 seconden lang zijn.
* De software moet met een smartphone te gebruiken zijn.
* De software moet een catalogus aankunnen van minimaal 2000 nummers.
* De gemiddelde zoektijd van een opgenomen fragment mag maximaal twee seconden zijn, waarvan minimaal 90% onder de drie seconden blijft.
* Bij een signaal-ruisverhouding vanaf 3dB en hoger moet er in minimaal 95% van de gevallen een match gevonden worden.
* Bij een signaal-ruisverhouding van 0 dB moet er in minimaal 80% van de gevallen een match gevonden worden.
* Bij een signaal-ruisverhouding van -3 dB moet er in minimaal 70% van de gevallen een match gevonden worden.
* Er mag in maximaal 5% van de gevallen een fout-positief optreden.

Theorie

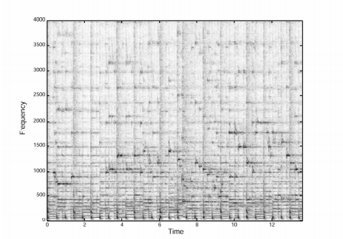
Algemeen

De techniek waarmee audiofragmenten worden herkend heet Audio Fingerprinting of Acoustic Fingerprinting (Wikipedia, 2018a). Men maakt dan een soort vingerafdruk van een klein deel van het audiofragment en vergelijkt dit volgens met vingerafdrukken van andere audiofragmenten in een database. Een veel gebruikte techniek hiervoor is het maken van een spectrogram van het audiofragment. In een spectrogram zijn met behulp van Fouriertransformaties de frequentie en de intensiteit van die frequentie uitgezet tegen de tijd. Vervolgens zijn er verschillende manieren om vanuit dit spectrogram tot een vingerafdruk te komen. Hieronder staan twee manieren om dit te doen. Eén van het bedrijf Shazam en één van het bedrijf Philips.

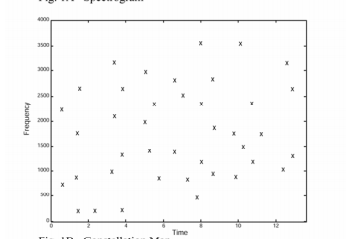
Shazam

Shazam maakt gebruik van de pieken in het spectrogram van een audiofragment om een match te kunnen maken tussen een deel en het origineel (Wang, 2003). Dit doen ze omdat pieken de grootste kans hebben om achtergrondgeluid en andere verstoringen te overleven. Als je alleen de pieken uit een spectrogram (afb 1) laat staan op het spectrogram, krijg je een soort sterrenkaart (afb 2). Vervolgens wordt elke piek één voor één als ankerpunt genomen. Deze ankerpunten hebben allemaal een eigen doelzone; een gebied op het spectrogram waarin het ankerpunt kan worden gekoppeld aan andere punten (afb 3). Vervolgens wordt er een hash [1] berekend van de frequenties van beide punten en hun onderlinge tijdsverschil (afb 4). Deze hash slaan ze vervolgens op in een database. Wanneer er een audiofragment gematcht moet worden maken ze op de zelfde manier vingerafdrukken en kijken of een vingerafdruk in de database voorkomt. Wanneer er meerdere matches zijn én wanneer de tijd tussen de vingerafdrukken voor een groot deel klopt, is er een match.

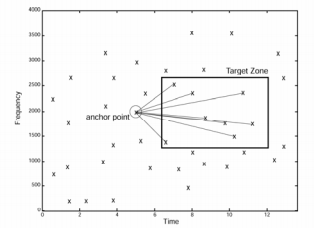
[1] Een hash: een functie die voor elke andere input een zo goed als unieke output geeft.



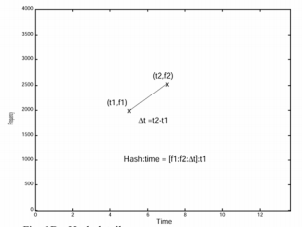
Afb 1



Afb 2



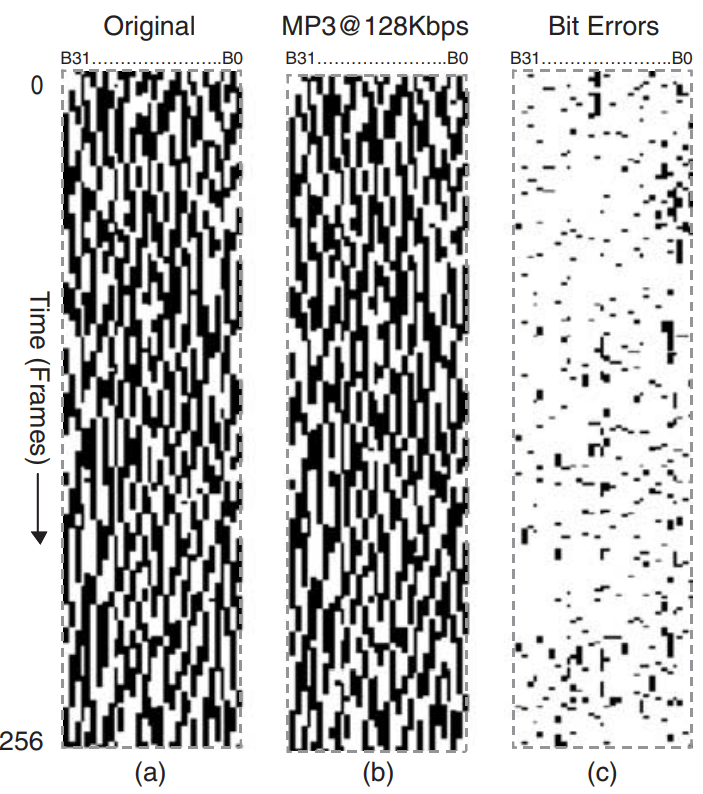
Afb 3



Afb 4

Philips

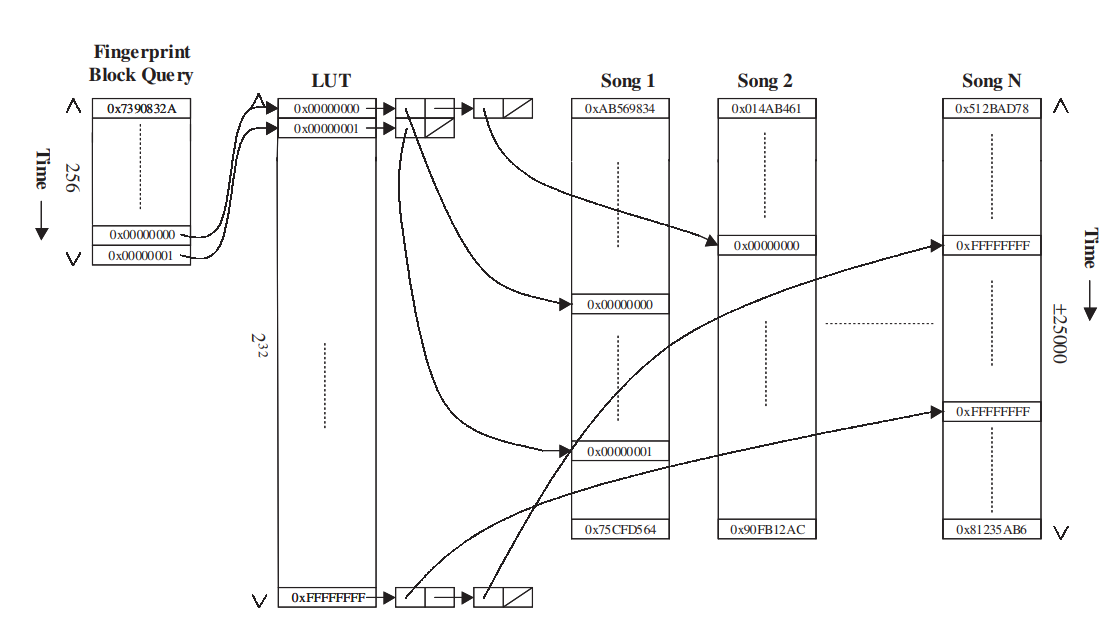
Ook Philips gebruikt spectrogrammen voor het herkennen van audiofragmenten (Haitsma & Kalker, 2002). Een audiofragment wordt opgedeeld in fragmenten van drie seconden; een vingerafdruk blok. Elk blok wordt vervolgens opgesplitst in 256 delen van elk 11.6 milliseconden. Van elk van deze delen wordt een spectrogram gemaakt. Vervolgens worden tussen 300 en 2000 Hz 33 frequentiebanden gekozen die logaritmisch verdeeld liggen. Vervolgens gebruiken ze onderstaande functie , waarin het tijdsfragment is, de frequentieband en de energie van de frequentieband in tijdsfragment is, om te bepalen of de frequentieband in het tijdsfragment een waarde van 1 of van 0 zal krijgen. Van de 32 uitkomsten maken ze vervolgens een 32-bits sub-vingerafdruk. Als ze 256 sub-vingerafdrukken bij elkaar zetten, krijgen ze een vingerafdrukblok (afb 5a). Als ze dit vergelijken met hetzelfde fragment, maar dan in MP3 formaat (afb 5b), krijgen ze een diagram waarin de verschillen te zien zijn (afb 5c).



Afb 5

Wanneer het aantal verschillen onder de grenswaarde ligt, worden de twee fragmenten beschouwd als een match.

In afbeelding 6 is te zien hoe er door de database gezocht wordt naar een match. Er wordt vanuit gegaan dat tenminste één sub-vingerafdruk identiek zal zijn aan het origineel in de database. Eerst wordt van alle mogelijke sub-vingerafdrukken een lookup-table gemaakt, een soort database waarin wordt verwezen naar de plek van de sub-vingerafdruk in de volledige database. Eerst wordt de laatst gegenereerde vingerafdruk opgezocht in de lookup-table. Deze verwijst naar een lied in de database. Vervolgens wordt de gehele vingerafdruk vergeleken met het lied in de volledige database. Wanneer het aantal verschillen beneden de grenswaarde ligt, is het een match. Wanneer dit niet zo is wordt verder gezocht met de op één na laatst gegenereerde sub-vingerafdruk en wanneer ook dit geen match oplevert wordt de op twee na laatst gegenereerde sub-vingerafdruk gebruikt, enzovoorts. Als alle sub-vingerafdrukken getest zijn, maar geen van de sub-vingerafdrukken een aantal verschillen opleverde dat lager was dan de grenswaarde, wordt er besloten dat er geen match gevonden kan worden.



Afb 6

Plan van aanpak

Er zijn een aantal onderdelen waarvoor een strategie bedacht moet worden alvorens aan het ontwerpen te beginnen. Die onderdelen zijn de back-end (de server) en hoe deze fragmenten matcht, de front-end (de app), communicatie tussen front-end en back-end, opslag en programmeertaal.

Back-end

In de back-end wordt de database opgeslagen met alle nummers, omdat het niet praktisch is om een enorme database van nummers met bijvoorbeeld een app mee te leveren. Bovendien kan zo de catalogus uitgebreid worden zonder dat alle gebruikers hun software moeten updaten.

Voor het matchen is besloten de manier van Shazam voor een groot deel te gebruiken. Deze manier staat namelijk toe een traditionele databasesoftware als MySQL te gebruiken in plaats van zelf databasesoftware te moeten maken wat bij de manier van Philips wel noodzakelijk is. Daarnaast kost de manier van Philips misschien te veel tijd om te maken binnen het gegeven tijdsbestek van 80 uur. Omdat het hashing algoritme van Shazam niet bekend is, wordt er gekozen voor het SHA-512 algoritme. Dit algoritme heeft 512 bits aan entropie, wat betekent dat er ongeveer mogelijke hashes zijn. Voor deze toepassing is dat meer dan genoeg.

Als besturingssysteem is de Linuxdistributie Ubuntu gekozen, want dit brengt geen extra kosten met zich mee en de schrijver heeft al ervaring met het beheren en configureren van Ubuntu servers.

Front-end

Als front-end wordt er een Android app gemaakt. Op deze manier is de software eenvoudig overal mee naartoe te nemen. Daarnaast is Android de marktleider voor besturingssystemen voor smartphones (Statista, 2018). Verder is er alleen beschikking tot een Android toestel en niet tot een iOS toestel om de app op te testen. Ook heeft de schrijver veel ervaring met het maken van apps voor Android en zo goed als geen ervaring met het maken van iOS apps.

Communicatie front-end en back-end

Om het geluidsfragment te kunnen matchen moet er data-uitwisseling zijn tussen de front-end en back-end. Er zijn hiervoor twee opties: het geluidsfragment wordt naar de server gestuurd en wordt daar geanalyseerd, of, het geluidsfragment wordt op de front-end geanalyseerd en de berekende hashes worden naar de server gestuurd voor vergelijking met de database. Beide methodes hebben voor- en nadelen, welke hieronder beschreven staan.

**Methode 1: geluidsfragment naar de server versturen**

Voordelen:

* De code voor het analyseren van de fragmenten hoeft maar één keer geschreven te worden.

Nadelen:

* Vergeleken met het verzenden van alleen de hashes gebruikt deze methode veel meer bandbreedte, wat de capaciteit van een server beperkt, de gebruiker meer geld kan kosten en waarvoor een beter netwerk nodig is.
* Het streamen van audio is ingewikkelder dan het versturen van een hash, omdat de hash in principe kale tekst is.
* Voor het efficiënt versturen van een geluidsfragment moet compressie toegepast worden, wat het herkennen van een fragment moeilijker maakt.

**Methode 2: hashes worden verstuurd naar de server**

Voordelen

* Het versturen van hashes gebruikt minder data dan het versturen van geluidsfragmenten.
* Het versturen van hashes heeft minder bandbreedte en een minder sterke verbinding nodig dan het versturen van een geluidsfragment.
* De server wordt minder belast omdat het meeste rekenwerk op de client plaats vindt.

Nadelen:

* De code voor het genereren van een hash moet twee keer gemaakt worden.
* Wanneer er twee programmeertalen gebruikt worden bestaat de kans dat hetzelfde audiofragment een ander resultaat oplevert, wat het matchen bemoeilijkt. Het is namelijk niet mogelijk om dezelfde libraries voor de app te gebruiken als voor de server. Daarnaast gaan verschillende programmeertalen ook anders om met bijvoorbeeld kommagetallen en delen door nul.

Ook al zijn er meer voordelen voor de tweede methode, er is toch gekozen voor de eerste methode, aangezien de kwestie dat het resultaat in beide talen hetzelfde moet zijn erg zwaar meetelt. Daarnaast bespaart het veel tijd wanneer de vingerafdruk-code slechts één keer gemaakt moet worden. Bovendien valt het argument dat het versturen van audio meer data en bandbreedte kost te verwaarlozen, aangezien er tegenwoordig vrijwel overal stabiele en snelle verbindingen beschikbaar zijn en niet-gecomprimeerde 16-bits 44.1kHz mono audiobestanden van tien seconden lang ongeveer 900kB groot zijn (Crowley, sd).

Opslag

De database op de server mag niet te groot worden. Dit zou namelijk meer hardware vereisen waardoor de kosten zouden toenemen en het zou langer duren om een match te vinden, aangezien er meer data doorzocht moet worden. Wanneer we uitgaan van ongeveer 50.000 vingerafdrukken per liedje en 2000 liedjes in de database, komen we voor enkel de hashes al uit op bits, oftewel ruim 51 Gb. Tel hierbij voor elk vingerafdruk nog een 16-bits integer voor het ID van het liedje bij op en je komt nog een paar Gb hoger uit. Dit is redelijk veel data. Als oplossing hiervoor kunnen we de grootte van de vingerafdruk verkleinen. Wanneer we bijvoorbeeld alleen de eerste 64 bits van de SHA-512 hash pakken verkleinen we de grootte van de database al met ongeveer een factor 8. We behouden nog steeds een entropie van , wat ruim genoeg is voor de toepassing. Aangezien het aantal vingerafdrukken per lied later groter of kleiner kan blijken, wordt er voor nu nog gekozen voor het niet inperken van de grootte van de vingerafdruk. Mocht later blijken dat de database te grote proporties aanneemt, dan kan er alsnog besloten worden dit wel te doen.

Programmeertaal:

Er moet een programmeertaal gekozen worden voor zowel de Android app als de server. Voor Android apps zijn er twee opties: Kotlin en Java. Aangezien de schrijver geen ervaring heeft met Kotlin, maar wel redelijk veel met Java, wordt er voor de Android app gekozen voor Java.

Voor de server zijn er tal van opties, waaronder C++, C#, Python, NodeJS, Java en PHP. De talen waar de schrijver het meeste ervaring mee heeft zijn Python, Java en PHP.

Python en Java hebben beide als voordeel dat er veel zogenoemde libraries beschikbaar zijn. Dit zijn stukken code die door andere mensen zijn geschreven om andere programmeurs tijd te besparen. Vaak werken er veel mensen aan een library, wat ervoor zorgt dat de kans op bugs vaak kleiner is dan wanneer je zelf hetzelfde zou maken.

Ten opzichte van Python en PHP heeft Java als nadeel dat om veranderingen te kunnen testen, je heel het programma moet compileren. Compileren wil zeggen dat er van de code machinetaal wordt gemaakt die de processor kan begrijpen. PHP en Python doen dit allebei terwijl ze draaien, waardoor je een wijziging meteen kunt testen zonder te hoeven wachten op de compiler.

Ten opzichte van PHP hebben Python en Java een voorsprong op het gebied van parallel werken. Java en Python hebben namelijk allebei threads, Dit betekent dat Python en Java beter gebruik kunnen maken van processors met meerdere cores.

Afgaand op bovengenoemde voor- en nadelen is er daarom gekozen om Python te gebruiken op te server.

Planning:

|  |  |
| --- | --- |
| Datum | Wat af |
| 1 oktober 2018 | Plan van aanpak |
| 15 Oktober 2018 | Server-side fingerprinting |
| 22 oktober 2018 | Testen en bijstellen fingerprinting algoritme op grote schaal |
| 29 oktober 2018 | Server-side API |
| 12 NOVEMBER 2018 | Client API-integratie |
| 19 NOVEMBER 2018 | Testen algoritme met achtergrondgeluid |
| 26 NOVEMBER 2018 | Verslag afronden |

Stappenplan

Hieronder staat welke stappen er moeten worden doorlopen voor zowel het opslaan als het matchen van audiofragmenten.

### Opslaan:

Fourier Transformatie 🡪 Pieken zoeken 🡪 Hash maken van pieken 🡪 hashes opslaan in database samen met bijbehorende tijd en id

### Matchen:

Opname 🡪 Uploaden opname naar server 🡪 Fourier Transformatie 🡪 Pieken zoeken 🡪 Hash maken van pieken 🡪 hashes zoeken in database 🡪 match bepalen doormiddel van meeste overeenkomsten in tijdverschil tussen hashes uit het origineel en hashes uit het fragment

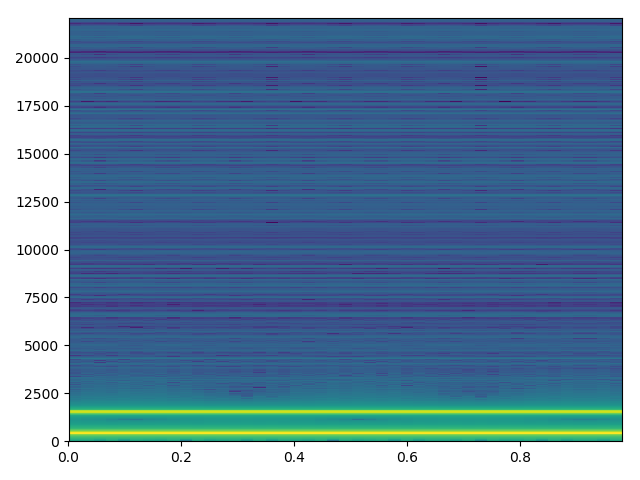
Ontwerpproces

Onderdeel 1: Fouriertransformaties:

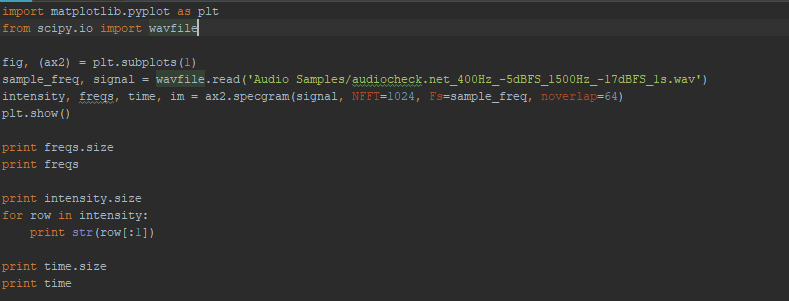
Als eerste moet het vingerafdruksysteem voor de server gemaakt worden. Als eerste moeten er hiervoor Fouriertransformaties toegepast worden, oftewel: er moet een spectrogram gemaakt worden. Na wat zoeken in de documentatie van verschillende bekende wiskundige python libraries, namelijk NumPy, SciPi en matplotlib, kwamen twee functies tevoorschijn die hierbij nuttig zijn: scipy.io.wavfile.read en matplotlib.pyplot.specgram.

**scipy.io.wavfile.read:**Met deze functie kan een WAVE audiobestand worden ingeladen als een array van waarden (SciPy, 2018). Deze array kan vervolgens gebruikt worden om Fouriertransformaties op toe te passen en zo een spectrogram te maken. De functie met argumenten luidt: scipy.io.wavfile.read(filename, mmap=False). Filename is hierin een relatief of absoluut pad naar het WAVE bestand dat moet worden gelezen. mmap is de optie of het bestand als memory-mapped moet worden geladen in het RAM geheugen met een standaard waarde van False. De functie geeft twee variabelen als output: rate (int) en data (numpy array). rate is een integer en geeft de bemonsteringsfrequentie waarmee het WAVE bestand gelezen is. data is de array van de samples (de waarde van een signaal op een bepaald tijdsinterval) van het WAVE bestand waarvan het type array hetzelfde soort items heeft als het WAVE-bestand.

**matplotlib.pyplot.specgram:**Met deze functie kan een spectrogram gemaakt worden van een signaal (matplotlib, 2018). Dit spectrogram kan vervolgens geanalyseerd worden op pieken om vingerafdrukken van te maken. De volledige functie met argumenten is:   
matplotlib.pyplot.specgram(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend\_none, window=mlab.window\_hanning, noverlap=128, cmap=None, xextent=None, pad\_to=None, sides='default', scale\_by\_freq=None, mode='default', scale='default', \*\*kwargs)  
Voor het meeste van de argumenten zijn de standaarden voldoende en er zal dan ook niet verder op worden ingegaan. De volgende argumenten zijn echter wel belangrijk: x is het signaal, in dit geval de data uit scipy.io.wavfile.read, NFFT is de blokgrootte in aantal datapunten die gebruikt wordt voor het berekenen van de Fouriertransformatie. Deze waarde moet voor efficiëntie een macht van twee zijn en er geldt hoe groter deze waarde, hoe minder rekentijd. Als laatste is het noverlap argument belangrijk. Het bepaalt hoeveel datapunten er moeten overlappen van het vorige blok bij het nieuwe blok. Dit argument heeft invloed op de nauwkeurigheid van het spectrogram. Later zal er met deze waarde worden getest om te kijken welke het beste resultaat levert voor de herkenning.  
De functie heeft vier variabelen als output: spectrum, freqs, t en im. spectrum is een tweedimensionale array (simpelweg een array van arrays met gelijke lengte) die de uitkomsten van de fouriertransformaties bevat. freqs is een array van de frequenties waarop getest is met de fouriertransformaties. t is een array van de tijdstippen in het midden van een blok waarop de fouriertransformatie is toegepast. im is de afbeelding van het spectrogram zelf.

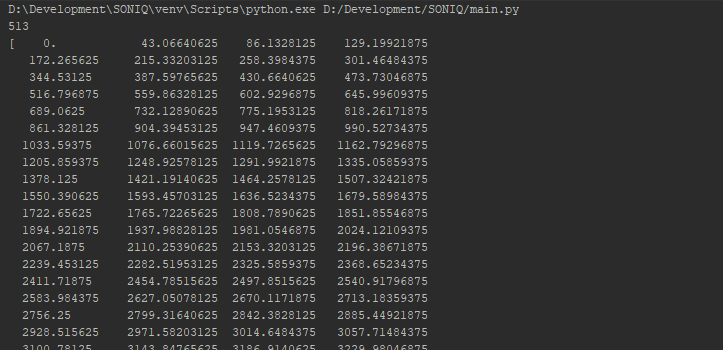
Eerst moet er uitgezocht worden of de gegevens afkomstig uit de functie specgram inderdaad de gegevens zijn die nodig zijn om vingerafdrukken uit te maken. Om dit te testen is er gebruik gemaakt van een testfragment waarin twee frequenties gebruikt zijn: 400Hz en 1500Hz. Het signaal van 400Hz is harder (-5dBFS) dan het signaal van 1500Hz (-17dBFS). Om te beginnen is er een spectrogram geplot met behulp van de functies scipy.io.wavfile.read en matplotlib.pyplot.specgram. Afbeelding 7 was daarvan het resultaat. Zoals te zien is zijn er twee felle gele lijnen rond de 400Hz en 1500Hz, zoals verwacht. De gebruikte functies zijn dus in staat het audiofragment juist te verwerken. Vervolgens moet worden uitgezocht of de spectrum variabele de gewenste data bevat. Hiervoor is het volgende stuk code gebruikt (afb 8):

Afb 7

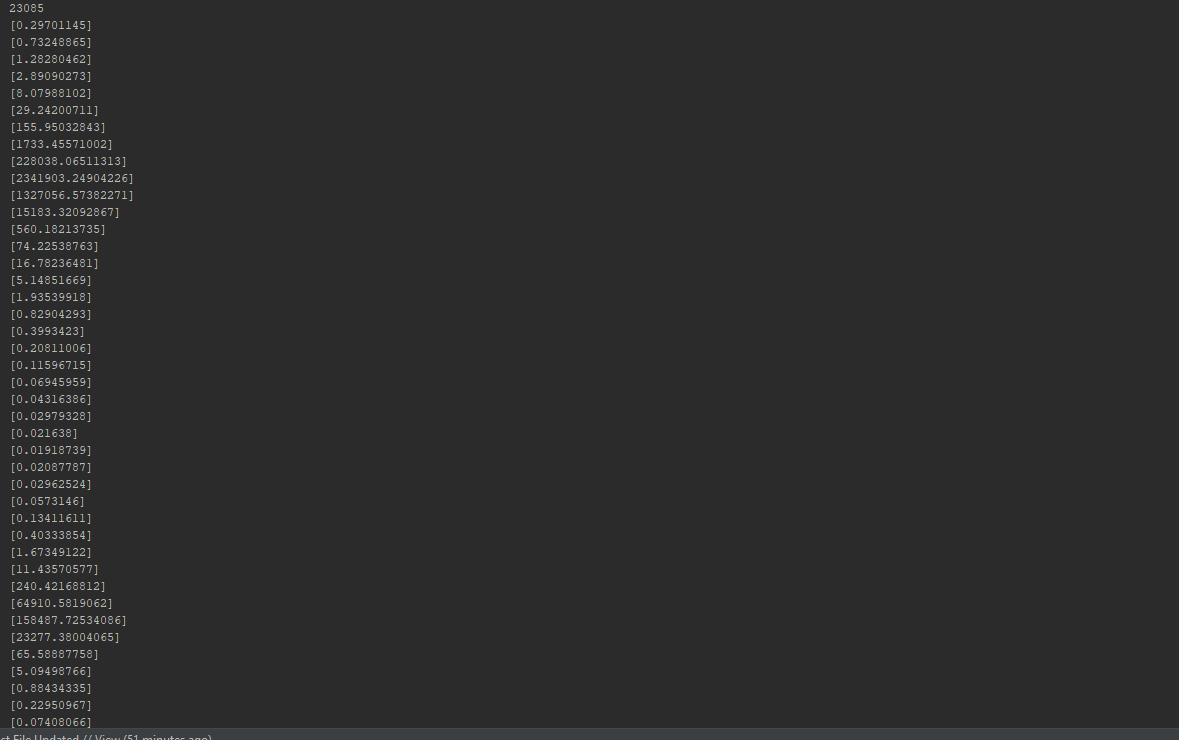


Afb 8

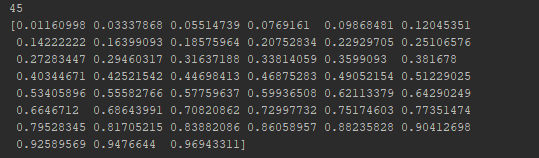
Hieronder zijn de beknopte resultaten gegeven van het stuk gebruikte code (afb 9,10,11):



Afb 9

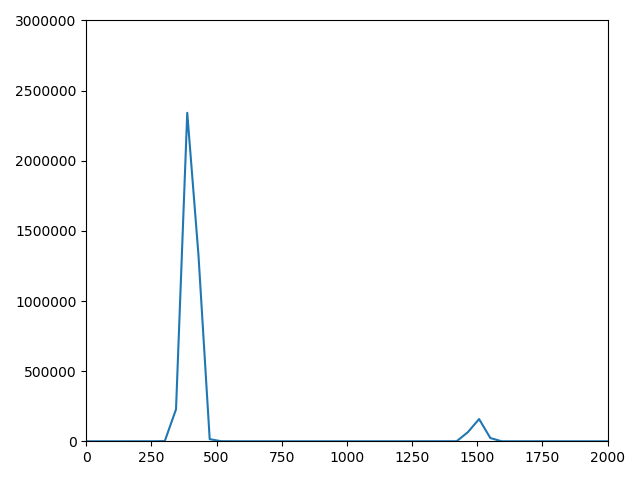
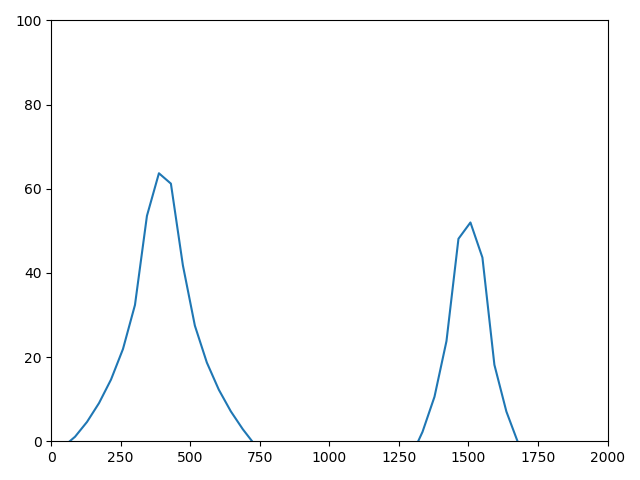


Afb 10



Afb 11

Zoals in afbeelding 10 te zien is, is de lengte van de intensity variabele (spectrum in de documentatie) 23085. De lengte van de freqs variabele is 513 (afb 9) en de lengte van time (t in de documentatie) is 45. Aangezien , bevat de spectrum variabele inderdaad de uitkomst van de fouriertransformatie voor een bepaalde frequentie op een tijdsinterval. Na het bekijken van de lengte van eerste array in de tweedimensionale array spectrum bleek dat deze 45 was. Dit betekent dat elke array alle uitkomsten van de fouriertransformatie op alle tijdstippen voor één frequentie bevat. Wanneer er van elke array in spectrum de eerste waarde gepakt wordt, krijgt men alle uitkomsten van de fouriertransformaties voor alle frequenties op één bepaald tijdstip (afb 10). Hierin is te zien dat er rondom de tiende en elfde waarde een piek zit. Wanneer er in de lijst van frequenties de tiende en elfde frequentie opgezocht wordt, blijken deze respectievelijk ongeveer 388Hz en 431Hz te zijn. Deze piek is dus van de 400Hz frequentie uit het audiofragment. Verderop bleek er ook nog een piek te zitten rond de 1500Hz, maar deze staat niet in de afbeelding. Vervolgens is de intensiteit per frequentie voor het eerste tijdsfragment geplot en daaruit kwam de grafiek uit afbeelding 12. Zoals te zien is de piek bij 400Hz vele malen groter dan die bij 1500Hz. Dit was ook de verwachte uitkomst was omdat het signaal van 400Hz ook sterker was dan dat van 1500Hz. Dit bewijst dat de gebruikte functies naast de juiste frequenties, ook goed onderscheid maken tussen de intensiteit van tonen. Voor de computer maakt dit grote verschil in getallen niet uit, maar om de grafieken duidelijker weer te geven is in afbeelding 13 dezelfde grafiek weergegeven, maar dan met logaritmische schaal.

Afb 12 Afb 13

Afgaand op het bovenstaande kan de conclusie getrokken worden dat spectrum inderdaad de juiste en benodigde data bevat om pieken uit te bepalen in de volgende stap.

Onderdeel 2: Vingerafdrukken maken

## Onderdeel 2.1: Pieken vinden

Nu de uitkomsten van de Fouriertransformaties bekend zijn moeten er in die uitkomsten pieken gevonden worden. Om dit te doen is een algoritme gemaakt dat bepaalt of een punt in zijn omliggende omgeving de hoogste waarde heeft. Een flow-chart hiervan is hieronder weergegeven (afb 14). Het valt wellicht op dat de pijl 14.1 niet de meest logische keuze is wat betreft optimalisatie. Immers, wanneer er een punt is gevonden dat hoger is dan het te onderzoeken punt, is het niet meer nodig de andere punten binnen de grenzen te onderzoeken, maar zou het sneller zijn meteen weer een nieuw punt te kiezen om te testen. Dit is gedaan omdat het niet mogelijk is om met normale for-else loops hierop te controleren zonder een extra variabele toe te voegen die bij elke herhaling gecontroleerd zou moeten worden. Dit zou dan weer voor tijdsverlies zorgen, dus er is gekozen om dit niet te doen.

JA

Kies het eerstvolgende punt (test\_punt) uit de test\_rij

Is test\_punt groter dan punt?

Piek toevoegen aan lijst met gevonden pieken

Is test\_punt het laatste punt in test\_rij binnen de gestelde grenzen?

Is test\_rij de laatste rij binnen de gestelde grenzen?

NEE

Kies het eerstvolgend punt (punt)

Kies de eerstvolgende nabij gelegen frequentierij binnen de gestelde grenzen (test\_rij)

JA

JA

NEE

NEE

*14.1*

Afb 14

## Onderdeel 2.2: Pieken combineren naar vingerdrukken

Nu de pieken uit het spectrogram gehaald zijn moeten ze gecombineerd worden tot een vingerafdruk. Omdat er hiervoor de array met pieken vaak doorlopen moet worden en dit veel tijd kost bij veel pieken is er hierom een index gemaakt van de lijst. De index is opgebouwd als een array van de indexen van het eerst voorkomende item met een bepaalde frequentie in de array met pieken. Wanneer er in de lijst met pieken een bepaalde frequentie niet voorkomt, wordt de index van de vorige frequentie gebruikt. Hieronder een voorbeeld van pieken en de bijbehorende index (merk hierbij op dat arrays beginnen met de index 0 en niet 1):

Pieken: [[1,23], [1,26], [3,98], [4,123], …]

Bijbehorende index: [0, 0, 0, 2, 3, …]

Door deze index te maken wordt er veel tijd bespaard tijdens het maken van vingerafdrukken aangezien het algoritme weet waar het moet beginnen met zoeken naar pieken binnen de gestelde frequentie grenzen. Hieronder is het algoritme voor het combineren van pieken tot een vingerafdruk vereenvoudigd gevisualiseerd (afb 15):

De test\_peak heeft een te kleine tijdwaarde

peak was de laatste van peak\_test\_row

Gebruikte variabelen:

Peaks [[0, 13], [2, 218], …] met index peak\_index=0 en tweede index peak\_test\_index=0  
Index [0, 0, 1, …] met index index\_index

Neem peaks[peak\_index] als peak

Neem peak\_test\_row binnen de frequentie grenzen en zoek daarmee start\_search in index. Neem stop\_search door het volgende getal uit index te pakken. Indien start\_search de laatste waarde uit index is, neem dan de lengte van pieken.

Kies test\_peak[peak\_test\_index] uit peaks met peak\_test\_index als de som van peak\_index en een waarde tussen start\_search en stop\_search

Ligt test\_peak op peak\_test\_row en valt deze binnen de gestelde tijdgrenswaarden?

*(de volgende functie is een vereenvoudigde versie van de werkelijke functie)* Gebruik  
**sha512(peak[0] + “-“ + peak\_test[0] + “-“ + (peak\_test[1] – peak[1]))**  
om een hash te maken en voeg deze samen met de tijd toe aan de lijst met vingerafdrukken.

peak was niet de laatste van peak\_test\_row

De test\_peak heeft een te grote tijdwaarde

JA

NEE

*afb 15*

Afb 15

Zoals te zien is wordt niet alleen de combinatie van de twee pieken gebruikt voor de vingerafdruk, maar ook het tijdstip van de eerste piek. Dit tijdstip hebben we namelijk later nodig om te controleren of een fragment inderdaad onderdeel is van het gehele fragment (Wang, 2003). Het resultaat van de vingerafdrukfunctie (afb 15) is dan ook een array bestaande uit de hash van de piekgegevens en het tijdstip van de eerste piek.

Onderdeel 3: Opslag van vingerafdrukken

Nu de vingerafdrukken gemaakt zijn, moeten ze worden opgeslagen. Hiervoor is MariaDB gebruikt, een populaire, verbeterde versie van MySQL. Het verschil tussen deze twee databasesystemen is erg klein, daarom volstaat de documentatie van MySQL in de meeste gevallen ook voor MariaDB. Binnen de database zijn twee tabellen nodig: één voor de opslag van de gegevens van nummers en één voor de opslag van vingerafdrukken. In de tabel van de nummers zijn vijf kolommen nodig, namelijk: id, bestandsnaam, titel, artiest en voltooid. In tabel 1 is de opbouw van deze tabel te zien. De eerste rij is de kolomnaam en de tweede rij het datatype (MySQL, sd) met bijbehorende argumenten.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Id | filename | title | artist | fingerprinted |
| INTEGER (32 bits) AUTO-INCREMENT zodat deze automatisch hoger wordt bij nieuwe toevoegingen. | **CHAR** met lengte 100 (100 bytes). CHAR is een string met vaste lengte, de lengte 100 is lang genoeg voor bestandsnamen | **CHAR** met lengte 100 (100 bytes). De titel van het nummer zal niet gauw langer dan 100 karakters zijn en is dus genoeg. | **CHAR** met lengte 100 (100 bytes). De artiest van het nummer zal niet gauw langer dan 100 karakters zijn en is dus genoeg. | **TINYINT**. Een TINYINT is een INTEGER met een lengte van 4 bits, de maximale waarde hierdoor is 255. Hij wordt hier gebruikt als boolean, dus hij is 0 of 1 |

Tabel 1

Voor de vingerafdrukken tabel zijn vier kolommen nodig, namelijk id, het id van het liedje, de vingerafdruk en het tijdstip. De opbouw van deze tabel is in tabel 2 weergegeven. De eerste rij is de kolomnaam en de tweede rij het datatype met bijbehorende argumenten.

|  |  |  |  |
| --- | --- | --- | --- |
| Id | song\_id | fingerprint | time |
| UNSIGNED INTEGER (32 bits integer) AUTO-INCREMENT zodat deze automatisch hoger wordt bij nieuwe toevoegingen. De waarde is maximaal 4.294.967.295. | **UNSIGNED SMALLINT** (16 bits integer). Hiervan is de waarde maximaal 65535, mochten er meer liedjes in de database komen, dan kan de lengte verlengd worden ten koste van opslag. | **CHAR** met lengte 128 (128 bytes). De SHA-512 hash is 128 tekens lang, dus dit is precies genoeg. | **MEDIUMINT**. (24 bits integer). De maximale waarde hiervan is 16777215. SMALLINT was in sommige gevallen te klein voor een aantal nummers, dus is er voor een MEDIUMINT gekozen. |

Tabel 2

Echter, na het indexen van 10 test nummers was de database 389.9 MiB groot, wat neerkomt op 78 GiB voor 2000 nummers. Dit is een zeer grote database voor dat aantal nummers. Er is daarom gekozen om het datatype van die kolom aan te passen naar een veld met 8 bytes. We houden hierdoor een entropie van 2^64, wat ruim genoeg is voor het opslaan van vingerafdrukken. Hierna was de database geslonken naar 54.1 MiB, wat resulteert in ongeveer 10.8 GiB voor 2000 nummers. Een verbetering van ongeveer 8 keer. De herziene versie van deze tabel is te zien in tabel 3.

|  |  |  |  |
| --- | --- | --- | --- |
| Id | song\_id | fingerprint | time |
| UNSIGNED INTEGER (32 bits integer) AUTO-INCREMENT zodat deze automatisch hoger wordt bij nieuwe toevoegingen. De waarde is maximaal 4.294.967.295. | **UNSIGNED SMALLINT** (16 bits integer). Hiervan is de waarde maximaal 65535, mochten er meer liedjes in de database komen, dan kan de lengte verlengd worden ten koste van opslag. | **BYTE** met lengte 8. Door de grootte van dit veld in te korten houden we een kleiner, maar nog steeds groot genoeg aantal mogelijkheden over en verkleinen we de database aanzienlijk. | **MEDIUMINT**. (24 bits integer). De maximale waarde hiervan is 16777215. SMALLINT was in sommige gevallen te klein voor een aantal nummers, dus is er voor een MEDIUMINT gekozen. |

Tabel 3

Maar er was nog een probleem: zoek tijd. Wanneer er in een database met 200 nummers gezocht werd naar één bepaalde vingerafdruk, duurde dit bijna 6 seconden. Dit is veel te lang om de gestelde eis van gemiddeld twee seconden te kunnen halen. Om dit op te lossen is er een index toegevoegd op het fingerprint veld van de fingerprints tabel. Het resultaat: de zoekopdracht duurde nu nog maar 0,000s in plaats van 6. Minder dan een milliseconde dus. De tijdswinst was dus minstens een factor 6000.

Onderdeel 4: Matchen fragment aan geheel

Om een fragment te matchen aan een nummer worden er op dezelfde manier als bij het origineel vingerafdrukken uit het geluidsfragment gehaald. Vervolgens moeten deze vingerafdrukken worden gezocht in de database. Eerst wordt er hiervoor een lijst gemaakt met het aantal matches per origineel. Hiervoor wordt de volgende query gebruikt:

SELECT song\_id, *COUNT*(\*) as cnt FROM (SELECT song\_id FROM fingerprints WHERE fingerprint IN (%s) GROUP BY song\_id,fingerprint) tmp GROUP BY song\_id ORDER BY cnt DESC

Zoals te zien is dit eigenlijk een query op een query. De binnenste query (SELECT song\_id FROM fingerprints WHERE fingerprint IN (%s) GROUP BY song\_id,fingerprint) wordt gebruikt om bij alle vingerafdrukken %s het bijbehorende nummer te vinden. De resultaten hiervan worden tijdelijk weggeschreven naar tabel tmp. Vervolgens wordt met de buitenste query het aantal keer geteld dat een nummer voorkomt in die tabel en wordt er gesorteerd op meeste overeenkomsten naar minste.

Daarna moet er berekend worden welk van de nummers de grootste kans heeft om ook daadwerkelijk het origineel te zijn. Om dit te doen wordt er een confidence waarde berekend. Hiervoor wordt per lied bekeken hoeveel overeenkomsten er zijn tussen het tijdstip van de vingerafdrukken van het fragment minus het tijdstip van de vingerafdruk van het origineel. Alleen tijdsverschillen die vaker dan één keer voorkomen worden meegenomen in de berekening. Om de confidence waarde te berekenen uit de tijdsverschillen is de volgende formule bedacht:

Het kwadraat wordt gebruikt zodat een groter aantal keer hetzelfde tijdsverschil ook een grote invloed heeft op de confidence waarde.

Wanneer de confidence waarde boven een bepaalde grenswaarde komt, is er sprake van een match. Deze grenswaarde zal later nog bepaald moeten worden tijdens het testen.

↪ Onderdeel 2.2: Pieken combineren naar vingerafdrukken

Toen het algoritme getest werd, bleek dat het erg inaccuraat was. Bij een test-database van 20 nummers werd maar in 30% van de gevallen het juiste antwoord gevonden. Dit kwam waarschijnlijk door een aantal ontwerpfouten in het algoritme voor het combineren van pieken tot een vingerafdruk (afb 15). Ten eerste werd het aantal frequenties rondom een piek beperkt waarin een tweede piek gezocht werd om te koppelen. Dit betekent dat er geen pieken van hoge frequenties gebruikt worden voor het maken van koppels met een piek van een bastoon. Daarnaast keek het koppelalgoritme ver vooruit in de tijd. Dit heeft als gevolg dat er veel koppels zijn waarvan de ene piek buiten het opgenomen fragment ligt en dus onbruikbaar is om mee te zoeken. Met deze aandachtspunten is vervolgens een nieuw algoritme gemaakt voor het koppelen van pieken (afb 16).

Gebruikte variabelen:

peaks [[0, 13], [2, 218], …] met index peak\_index=0 en tweede index peak\_test\_index=0  
timewindow

Valt peak2 binnen de gestelde tijdsgrens?

Kies peaks[peak\_index + i] als peak2

*(de volgende functie is een vereenvoudigde versie van de werkelijke functie)* Gebruik  
**sha512(peak[0] + “-“ + peak2[0] + “-“ + (peak2[1] – peak[1]))**  
om een hash te maken en voeg deze samen met de tijd toe aan de lijst met vingerafdrukken.

i = i +1

Sorteer peaks op de tijd-waarde van een piek

Kies peaks[peak\_index] als peak

Neem i = 1

nee

ja

Afb 16

Met dit nieuwe algoritme (afb 16) werden bij dezelfde 20 fragmenten alle fragmenten juist herkend. Dit algoritme is dus een stuk beter dan het andere en daarom is besloten om het algoritme afgebeeld in afbeelding 16 te gebruiken in plaats van het algoritme afgebeeld in afbeelding 15.

Onderdeel 5: Bepalen optimale instellingen

Nu het algoritme werkt, is het belangrijk optimale instellingen te vinden met betrekking op snelheid, accuraatheid en opslag. Voor het testen is er als volgt een test-set gemaakt: Met de microfoon van de laptop is een opname van tien seconden lang gemaakt van een telefoon die een liedje van YouTube afspeelde. Straalvorming en ruisonderdrukking stonden hierbij op de laptop uit en de speakeruitgang van de telefoon stond loodrecht op de microfoon van de laptop. De afstand tussen de telefoon en microfoon was ongeveer 30 centimeter. Er bevonden zich 200 nummers in de database.

Hieronder (tabel 4) staan de resultaten vermeld van een test-set dat 20 fragmenten bevatte waarvan het origineel zich in de database bevond en 5 fragmenten waarvan het origineel niet in de database stond. Zoals te zien is werd bij alle fragmenten een juiste uitkomst gegeven. De confidence waarden zijn erg variërend. Na een aantal fragmenten te hebben beluisterd met een lage waarde en een aantal met een hoge waarde, leek het erop dat het afspeelvolume hier vooral effect op had. Fragmenten met een lage confidence waarde waren doorgaans namelijk zachter dan die met een hogere waarde.

|  |  |  |  |
| --- | --- | --- | --- |
| Testopname nr. | Match/geen match | Confidence | Juist? |
| 0000-0 | Nee | - | Ja |
| 0000-1 | Nee | - | Ja |
| 0000-2 | Nee | - | Ja |
| 0000-3 | Nee | - | Ja |
| 0000-4 | Nee | - | Ja |
| 0002 | Ja | 3.544000 | Ja |
| 0007 | Ja | 71.753676 | Ja |
| 0017 | Ja | 0.901408 | Ja |
| 0023 | Ja | 49.849558 | Ja |
| 0039 | Ja | 3.445378 | Ja |
| 0059 | Ja | 37.162162 | Ja |
| 0066 | Ja | 4.333333 | Ja |
| 0068 | Ja | 18.670330 | Ja |
| 0081 | Ja | 42.137079 | Ja |
| 0100 | Ja | 0.948586 | Ja |
| 0109 | Ja | 1.595588 | Ja |
| 0116 | Ja | 35.753049 | Ja |
| 0131 | Ja | 1.831461 | Ja |
| 0132 | Ja | 1.388889 | Ja |
| 0143 | Ja | 0.457143 | Ja |
| 0149 | Ja | 3.006757 | Ja |
| 0167 | Ja | 11.276923 | Ja |
| 0175 | Ja | 37.901515 | Ja |
| 0179 | Ja | 12.044693 | Ja |
| 0200 | Ja | 29.332685 | Ja |

Tabel 4

Omdat het berekenen van de confidence waarde voor iedere mogelijke match relatief veel tijd kost, omdat voor elk lied tijdverschillen moeten worden opgezocht, is het belangrijk dat hierop bespaard kan worden. Als eerste is er geprobeerd om alleen voor de tien mogelijke matches met het hoogste aantal overeenkomsten in vingerafdrukken de confidence te berekenen. Dit scheelde gemiddeld tientallen seconden in tijd voor elke match en had verder geen invloed op de accuraatheid en confidence-waarde van een match.

Een tweede optimalisatie was het afkappen van de zoektocht naar een match wanneer een gevonden match hoogstwaarschijnlijk de sterkste match zou zijn. Dit is het geval wanneer de confidence waarde van een match boven een bepaalde grenswaarde uitkomt. Om de optimale grenswaarde te bepalen is er met verschillende grenswaarden getest. Uit de testresultaten bleek dat wanneer de grenswaarde onder de 5 kwam dit een negatieve invloed op de nauwkeurigheid had. Om te zorgen dat de grenswaarde niet alleen voldoende is voor deze test set en misschien niet zou voldoen voor andere test-sets, is er gekozen voor een minimale grenswaarde van 10. Deze aanpassing bood ongeveer een seconde tijdswinst bij matches waar de confidence boven deze grenswaarde uitkwam.

Vervolgens is er met de tijdmarge van de vingerafdrukken geëxperimenteerd. Voor deze tests zijn 300 liedjes geïmporteerd en werd een test-set van 100 fragmenten gebruikt. Deze fragmenten zijn op dezelfde manier gemaakt als bij de vorige test. Deze tijdsmarge bepaalt in hoeveel tijdsblokken twee pieken van elkaar mogen liggen om tot een paar gemaakt te worden. Een hogere waarde betekent dus een hoger aantal vingerafdrukken. Omdat het importeren van 300 nummers in de database al gauw drie uur of meer duurt, is er vanwege tijdsoverwegingen getest met intervallen van vijf seconden in plaats van een lager interval om het aantal tests te verminderen. Om te beginnen is er met een waarde van 20 tijdsblokken getest. Dit leverde een database van 4.1 GiB op voor 300 nummers en de volgende testresultaten:

========================================================================  
 Tests gedraaid: 100  
 Tests gematcht: 96  
 Tests correct: 95  
------------------------------------------------------------------------  
 Gemiddelde tijd: 1.66290s  
 90-percentiel: 2.963000s  
 Percentage correct van gematcht: 98.958333  
 Percentage correct van totaal: 95.000000  
========================================================================

Deze test is vervolgens nog drie keer herhaald om te zien of het andere resultaten gaf. De nauwkeurigheid was bij alle drie de tests hetzelfde, en de tijden verschilden enkele tientallen milliseconden van elkaar. In deze configuratie voldoet het tot nu toe aan de eisen voor tijd en nauwkeurigheid (zonder rekening te houden met de eisen voor verschillende hoeveelheden achtergrondgeluid).

Vervolgens is er met een waarde van 15 tijdsblokken getest. Ook deze test is herhaald en ook hier bleef de nauwkeurigheid door de tests heen hetzelfde en scheelde de tijd enkele tientallen tot honderd milliseconden van elkaar. Deze test gaf de volgende resultaten:

========================================================================  
 Tests gedraaid: 100  
 Tests gematcht: 97  
 Tests correct: 96  
------------------------------------------------------------------------  
 Gemiddelde tijd: 1.485126s  
 90-percentiel: 2.668954s  
 Percentage correct van gematcht: 98.969072  
 Percentage correct van totaal: 96.000000  
========================================================================

Uit deze resultaten blijkt dat zowel de tijd als de nauwkeurigheid erop vooruitgaan. Als mogelijke verklaring voor de lagere gemiddelde tijd is dat er minder vingerafdrukken gemaakt worden en er dus ook minder vingerafdrukken moeten worden opgezocht. De vooruitgang van de nauwkeurigheid is lastiger te verklaren, maar een mogelijke verklaring is het feit dat doordat het aantal vingerafdrukken kleiner wordt, er ook een kleinere kans is op een hash-collision. Een hash-collision treedt op wanneer twee verschillende inputs van een hash-functie dezelfde output geven. Door minder verschillende vingerafdrukken te hebben is de kans op zo’n collision ook kleiner. Ook was de database 3.3 GiB, een besparing van 20% tegenover een tijdmarge van 20 blokken.

De volgende test was met een marge van 10 blokken. Ook deze test is herhaald en hier bleef de nauwkeurigheid ook door de tests heen hetzelfde en scheelde de tijd ongeveer 100 tot 200 milliseconden van elkaar. De resultaten waren als volgt:

========================================================================  
 Tests gedraaid: 100  
 Tests gematcht: 99  
 Tests correct: 97  
------------------------------------------------------------------------  
 Gemiddelde tijd: 1.409510  
 90-percentiel: 2.226000s  
 Percentage correct van gematcht: 97.979798  
 Percentage correct van totaal: 97.000000  
========================================================================

Zoals te zien is er een vooruitgang op de totale nauwkeurigheid ten opzicht van zowel de eerste als de tweede test. Bij de nauwkeurigheid van de gematchte fragmenten is er echter een achteruitgang te zien. Op het gebied van tijd is de grootste vooruitgang te zien: ongeveer 0.1 seconde bij het gemiddelde en een daling van ongeveer 0.4 seconden bij het 90-percentiel. Dit brengt deze tijden ver onder de vereiste 2 en respectievelijk 3 seconden. Verder was de database nog maar 2.3GiB groot, een verbetering van 44 procent ten opzichte van de test met tijdsmarge 20.

Bij de laatste test met een testwaarde van vijf waren de resultaten dermate teleurstellend dat deze optie sowieso verviel. Alhoewel de database nog kleiner werd, namelijk 1.4GiB, bleef de tijd ongeveer hetzelfde, maar de nauwkeurigheid ging hard achteruit.

========================================================================  
 Tests gedraaid: 100  
 Tests gematcht: 69  
 Tests correct: 58  
------------------------------------------------------------------------  
 Gemiddelde tijd: 1.398584  
 90-percentiel: 2.128000s  
 Percentage correct van gematcht: 84.057971  
 Percentage correct van totaal: 58.000000  
========================================================================

Een totale nauwkeurigheid van 58 procent ligt ver beneden de eis. Ook al is niet bekend wat de signaal-ruisverhouding precies was, de muziek was wel duidelijk te horen in de fragmenten. Zelfs als alle ruis even sterk zou zijn als de muziek wordt er aan de eis van 80% herkenning niet voldaan.

Uit bovenstaande tests is bepaald dat een tijdsmarge van 10 de beste balans geeft tussen opslag, nauwkeurigheid en snelheid. Deze waarde zal dan ook vanaf nu gebruikt worden.

Andere waarden die we nog konden veranderen zijn de blokgrootte voor de fouriertransformaties, de overlap hiervan en de marges voor het vinden van pieken. Vanwege tijdsoverwegingen en het feit dat de resultaten meer dan voldoende zijn is besloten deze waarden verder niet te veranderen en te testen. Daarnaast staat in een aantal bronnen (Wang, 2003) (Yang, 2001) dat er een overlap van 50% gebruikt is, deze waarde wordt dan ook niet aangepast. Uiteindelijk zijn er dus de volgende aanpassingen gedaan aan de instellingen van het algoritme.

# Vingerafdrukken  
FINGERPRINT\_TIME\_WINDOW = 20 -> 10  
# Matchen:  
MIN\_CONFIDENCE\_FOR\_STOP = [niks] -> 10

Onderdeel 5: Server-side API

Na het testen is de server-side API ontworpen. Via deze API kan straks de cliënt resultaten krijgen voor een zoekopdracht. Er was de keuze tussen zelf een protocol ontwerpen of een bestaand protocol als HTTP te gebruiken. Zowel een eigen protocol als HTTP zijn gemakkelijk te implementeren in Python, maar na wat onderzoek te hebben gedaan bleek dat het maken van TCP/UDP verbindingen in Java ingewikkeld was en hier niet veel bestaande libraries voor zijn. Omdat de methode met een eigen protocol daardoor waarschijnlijk veel tijd ging kosten en er nog maar beperkte tijd over was, is er gekozen om HTTP te gebruiken als protocol tussen de client en de server.

De client zal om de seconde een nieuw deel van een fragment versturen totdat deze 10 seconden bereikt heeft. Om bandbreedte te besparen zal niet telkens opnieuw het gehele fragment verzonden worden, maar alleen de laatste seconde. De server heeft echter wel het gehele fragment nodig om het zo accuraat mogelijk te kunnen matchen. De server moet daarom de delen van elk fragment tijdelijk opslaan en op zo’n manier dat de juiste fragmenten aan elkaar gekoppeld kunnen worden. Hiervoor zal elke opdracht een uniek ID krijgen, wat ervoor zorgt dat fragmenten juist op de server kunnen worden opgeslagen zonder dat zoekopdrachten door elkaar heen gemixt kunnen worden. Hieronder (afb. 17) is een flowchart te zien van hoe dit in zijn werk gaat. Niet belangrijke HTTP headers zijn weggelaten.

Afb 17

🡨 *Zoekopdracht ID*

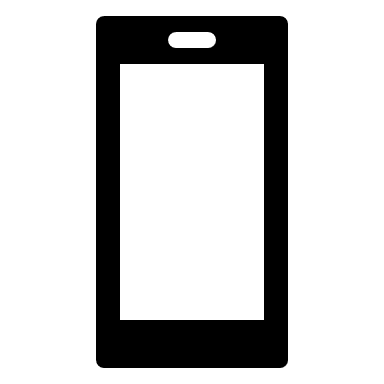
POST /search/data 🡪  
SEARCH\_ID = *Zoekopdracht ID*

*JSON array van opname*

GET /search/new 🡪



🡨 *JSON array met gevonden matches*



...

POST /search/data 🡪  
SEARCH\_ID = *Zoekopdracht ID*

*JSON array van opname*

🡨 *JSON array met gevonden matches*

Om te beginnen vraagt de client (links) met een GET verzoek een nieuw ID aan voor een zoekopdracht bij de server (rechts). Vervolgens zal de client de opname versturen door middel van een POST verzoek in de vorm van een JSON array. Bij dit verzoek zit tevens de SEARCH\_ID header die het ID van de zoekopdracht bevat. De server zal hierop reageren met de gevonden matches. Zolang de confidence-waarde niet boven de grenswaarde uitkomt, zal de client het POST-verzoek blijven sturen totdat dit wel het geval is of wanneer de tien seconden bereikt zijn. Wanneer er geen geldige SEARCH\_ID wordt opgegeven zal de server reageren met een 403-forbidden foutcode. Om de opname te versturen wordt er een JSON array gemaakt van de opname. Wanneer deze JSON data ongeldig is zal de server reageren met een 400-bad request code. In de andere gevallen zal de server reageren met een 200-OK code en de verzochte data.

Voor het maken van de server-side API zal de Flask library gebruikt worden (Ronancher, 2018). Deze library is gemaakt om eenvoudig API’s op te kunnen stellen. Als tijdelijke opslag van fragmenten zullen bestanden gebruikt worden met dezelfde naam als het ID van de zoekopdracht. In dit bestand zullen alle verzonden JSON arrays samengevoegd opgeslagen zijn. Elke minuut zullen oude bestanden opgeruimd worden die niet meer nodig zijn.

Afbeelding met schermafbeelding, monitor

Automatisch gegenereerde beschrijvingOnderdeel 6: client

De Android app heeft de volgende taken: audio opnemen, audio uploaden naar de server en het resultaat laten zien.

### Onderdeel 6.1: Gebruikersomgeving

Er is geprobeerd de gebruikersomgeving simpel en modern te houden. Er is daarom gekozen voor een enkele knop die het matchen start en een tekstveld om de URL van de backend op te geven. Zodra er een match gevonden is laat de app dit zien door middel van een info-kaart (afb 18). Om de app een moderne uitstraling te geven zijn ook nog enkele animaties toegevoegd.

### Onderdeel 6.2: Geluid opnemen

Omdat de server alleen ruwe data accepteert en dus geen MP3 of een andere codering is het opnemen van geluid redelijk eenvoudig. Met behulp van de AudioRecord functies die in Android ingebouwd zitten kan geluid met de juiste bemonsteringsfrequentie worden opgenomen (Google, 2018b). Voordat geluid kan worden opgenomen moet er een AudioRecord instance gemaakt worden. Dit kan met de volgende functie:   
public AudioRecord (int audioSource, int sampleRateInHz, int channelConfig, int audioFormat, int bufferSizeInBytes).

Afb 18

De audioSource is de audiobron waarvan wordt opgenomen. Aangezien er met de microfoon van het toestel moet worden opgenomen gebruiken we MediaRecorder.AudioSource.MIC (Google, 2018c). sampleRateInHz is de bemonsteringsfrequentie in Hertz. Aangezien 44100Hz de enige waarde is die gegarandeerd op alle toestellen werkt, wordt er hier de waarde 44100 gebruikt. channelConfig is het aantal kanalen dat gebruikt wordt voor de opname. Er is keuze uit mono en stereo. Aangezien de server maar een kanaal accepteert wordt er hier voor AudioFormat.CHANNEL\_IN\_MONO gekozen. audioFormat is de codering van de opgenomen audio. Er moet ruwe data naar de server verzonden worden, en omdat de compatibiliteit zo groot mogelijk moet zijn, is er gekozen voor AudioFormat.ENCODING\_PCM\_16BIT, aangezien dit de enige ruwe codering is die door alle toestellen ondersteund wordt (Google, 2018a). Het laatste argument, bufferSizeInBytes, moet worden berekend met de functie:  
public static int getMinBufferSize (int sampleRateInHz, int channelConfig, int audioFormat)

Hiervoor worden dezelfde waarden gebruikt als voor de AudioRecord instance. De integer die hier vervolgens uitkomt kan worden gebruikt in de AudioRecord(...) functie. Na het aanmaken van de AudioRecord instance, kan de opname gestart worden met audioRecord.startRecording() (Google, 2018b) en vervolgens kan de data worden uitgelezen met:  
public int read (short[] audioData, int offsetInShorts, int sizeInShorts)   
  
audioData is een array waarnaar de opgenomen data toegeschreven wordt beginnend bij de index offsetInShorts en een lengte van sizeInShorts. sizeInShorts kan berekend worden door de bemonsteringsfrequentie te vermenigvuldigen met het aantal seconden dat moet worden opgenomen. Omdat de read(...) functie de thread waarop hij wordt uitgevoerd blokkeert is het belangrijk om deze functie op een andere thread dan de UI-thread uit te voeren om te voorkomen dat het beeld blijft hangen.

Na het verkrijgen van de data van een opname moet deze naar de server verstuurd worden. De gebruiker kan zelf zijn API-eindpunt kiezen (afb. 18) waarnaar de data verstuurd wordt. Voor het uploaden van de data wordt gebruik gemaakt van de OkHttp library (Square, sd) en voor het verwerken van JSON wordt de GSON library (Google, 2018d) gebruikt. Elke keer wanneer een fragment van een seconde lang is opgenomen wordt dit geüpload naar de server die dit samenvoegt met eventueel eerder geüploade fragmenten tot een maximum van 10 seconden. Na het opzoeken van het geüploade fragment in de database door de server krijgt de app een reactie terug met daarin gegevens over de match en het gevonden origineel. Met de GSON library wordt dit antwoord omgezet in een Java-object. Wanneer er een zekere match is wordt deze door de app weergegeven op het scherm. Zodra een match gevonden is stopt de app met zoeken en breekt alle nog openstaande verbindingen af.

Testen

Om te beslissen of het gemaakt ontwerp aan alle eisen voldoet is er een geautomatiseerde testmethode gemaakt. Deze testmethode bestaat uit twee delen: het genereren van testfragmenten en het bepalen van de nauwkeurigheid wanneer deze getest werden.

Testfragmenten genereren

Om de testfragmenten te genereren zijn er twee geluidsfragmenten nodig, namelijk het origineel en een fragment met ruis. Als ruis is gekozen voor een geluidsopname van de aula tijdens de middagpauze. Er is tijdens de middagpauze veel lawaai in de aula en omdat het een geluidsopname is, zit er ook ruis in verwerkt die zou ontstaan tijdens een opname. Er is voor een geluidsopname gekozen, omdat dit een beter beeld geeft over de werkelijkheid dan bijvoorbeeld witte ruis.

De testfragmenten moeten een bepaalde signaal-ruisverhouding hebben (SNR). Om te zorgen dat een testfragment een bepaalde SNR heeft moet de ruis versterkt of verzwakt worden afhankelijk van de sterkte van het origineel. Omdat de SNR in decibel wordt uitgedrukt moet er rekening worden gehouden met de logaritmische schaal van deze eenheid. Om deze vermenigvuldigingsfactor uit te rekenen worden onderstaande formules gebruikt:

[1] Er wordt hier gebruik gemaakt van absolute waarden omdat we de gemiddelde uitwijking willen weten, en deze kan zowel positief als negatief zijn

11Vervolgens kan het testfragment gemaakt worden door elke waarde van de te vermenigvuldigen met en de uitkomst hiervan op te tellen bij het origineel: .

Door vervolgens het gemaakte fragment op te slaan met de bestandsnaam [song\_id] [SNR].wav kan het programma later gemakkelijk uit de bestandsnaam halen wat het gewenste resultaat moet zijn en weet het welke SNR dit fragment heeft.

Testfragmenten testen

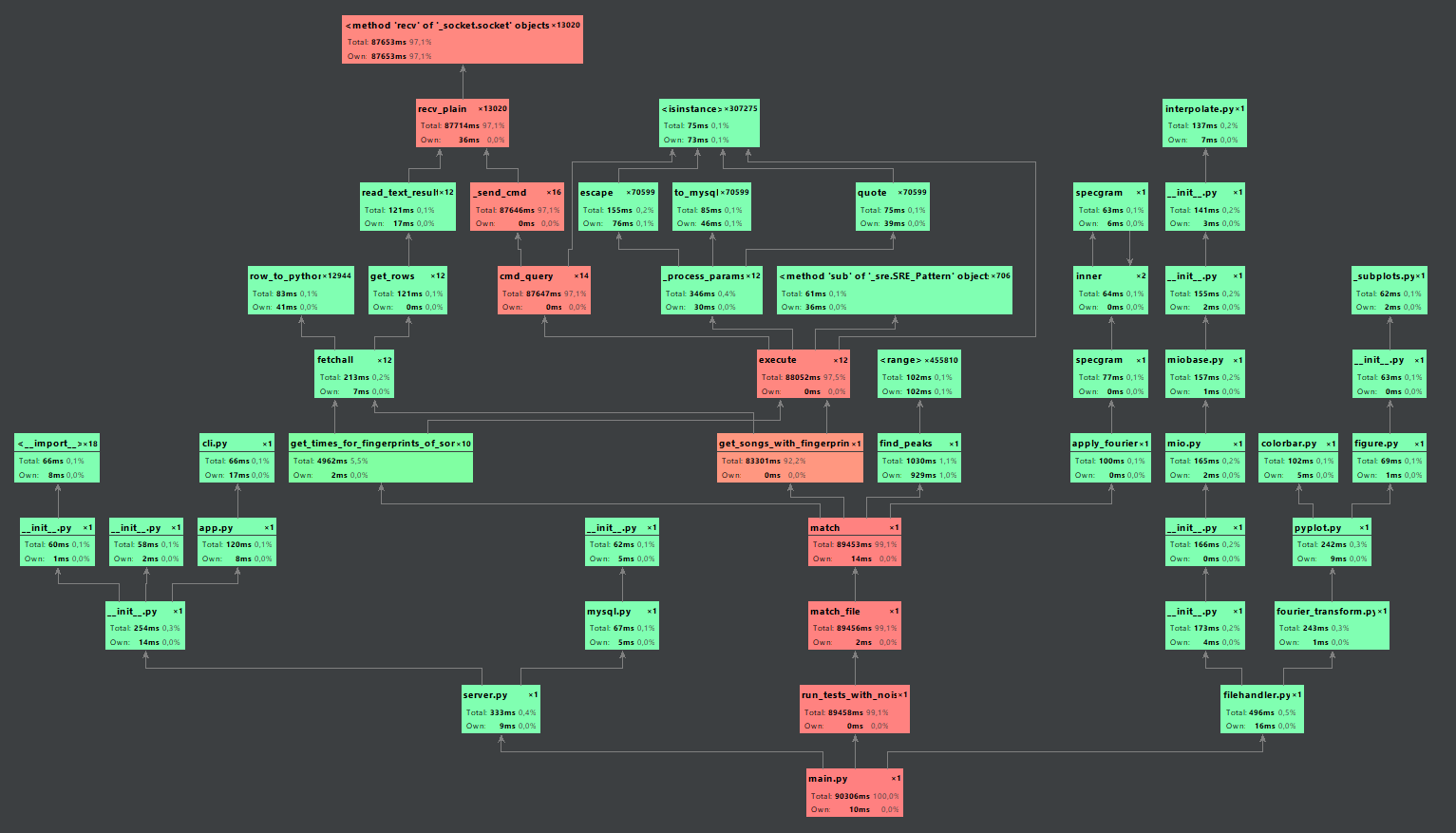
Om alle testfragmenten te testen moeten alle testfragmenten in dezelfde map geplaatst worden. Vervolgens kan het programma voor elk bestand het bijbehorende song\_id en de bijbehorende SNR vinden. De resultaten van de match worden vergeleken met het song\_id uit de bestandsnaam en er wordt vervolgens per SNR bepaald hoeveel van de tests succesvol waren.

Resultaten

Voor het testen is de Top 2000 van 2010 gebruikt. Deze liedjes werden allemaal geïmporteerd en voor 200 liedjes werden testfragmenten met een SNR van -3, 0, 3, en 6 dB aangemaakt. Vervolgens zijn deze fragmenten getest.

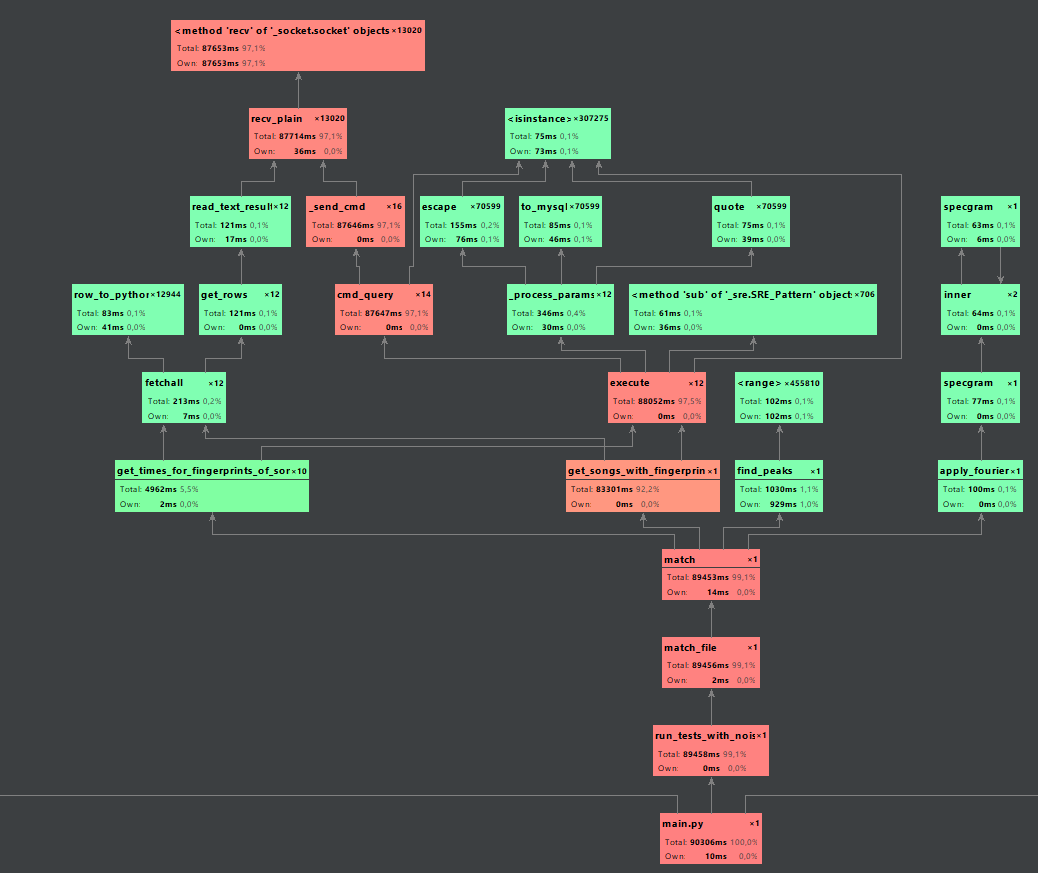
Zodra de tests werden aangezet viel één ding meteen op: het programma was traag. Na met een stopwatch te hebben gekeken hoe lang er over één test gedaan werd, bleek dit ongeveer 80 tot 90 seconden te zijn. Dit is veel te lang en als er bij elke test zolang zou moeten worden gewacht zou dit ongeveer 23 uur kosten. Omdat dit qua tijd niet rendabel is, is besloten te onderzoeken waarom dit zo lang duurt.

Onderzoeken traagheid

Om de traagheid te onderzoeken kan een zogenaamde profiler gedraaid worden. Een profiler houdt per functie bij hoe vaak deze wordt aangeroepen en hoe lang deze functie bezig is. Het draaien van een profiler gaf het volgende resultaat (afb 19):

Afb 19

In deze diagram (afb 19) is een kleurenschaal gebruikt. Hoe langer een functie nodig had in vergelijking met de totale tijd, hoe roder deze is. Zoals te zien zijn er een aantal blokken rood/oranje. Het onderste blok (main.py) is het bestand van waaruit alles wordt uitgevoerd, dus het is logisch dat deze erg donkerrood is. Om een indruk te krijgen van wat er zo lang duurt, is een bepaald gedeelte van afbeelding 19 belangrijk. Dit gedeelte is hieronder (afb 20) weergegeven:



Afb 20

Het matchen van één fragment duurde in dit geval 89456 ms (match\_file). Hiervan duurde de functie get\_songs\_with\_fingerprints het langste (83301 ms). Ook duurde het uitvoeren van get\_times\_for\_fingerprints\_of\_song relatief lang (4962 ms). Deze twee functies roepen de database aan. Blijkbaar heeft de database dus moeite met het vinden van data. Samen waren deze twee functies verantwoordelijk voor ongeveer 98,7% van de zoektijd. Het probleem lijkt dus vooral in de database software te liggen.

Met 2000 nummers in de database bevinden zich ongeveer 175 miljoen rijen in de database. Dit komt neer op een grootte van 16.7 GiB. Dit is natuurlijk een enorme hoeveelheid data om snel te kunnen doorzoeken en daarom zijn er eerder ook al indices toegevoegd zodat niet telkens de gehele 16.7 GiB hoeft worden doorzocht, maar er sneller kan worden opgezocht waar een rij met een bepaalde vingerafdruk zich bevindt.

Om te kijken of de traagheid misschien te wijten was aan een grote hoeveelheid duplicaten, is er een tweede database aangemaakt die precies dezelfde opzet heeft als in tabel 1 en 3, maar waarvan de fingerprint kolom nu 10 bytes lang is in plaats van 8. Dit zorgt voor ongeveer 65.000 keer zo veel mogelijkheden waardoor het aantal duplicaten aanzienlijk zou moeten dalen.

Na het opnieuw importeren van dezelfde 2000 nummers bleek dit geen verbetering met zich mee te brengen, zoals te zien in afbeelding 21. De database-functies nemen samen ongeveer 84 seconden in beslag, wat neerkomt op 98,7% van de totale tijd. Wel was er een verschil in de grootte van de database: met 17,9 GiB is deze database ongeveer 7% groter dan de database voor de aanpassing van de fingerprint kolom. Omdat de aanpassing dus alleen meer opslagruimte in beslag neemt en verder geen merkbare verbetering met zich meebrengt is besloten deze verandering ongedaan te maken.

Afbeelding met schermafbeelding, tekst

Automatisch gegenereerde beschrijving

Afb 21

Omdat de database wel snel was tijdens de eerste test met 300 nummers in de database bestaat het vermoeden dat de traagheid is ontstaan door de grote hoeveelheid data in één database. In MariaDB is het mogelijk om een database op te delen in kleinere delen. Dit kan door partities te maken. Deze partities kun je baseren op de waarde van een bepaald veld. Om te testen of dit effect heeft op de zoektijd zijn 16 partities gemaakt gebaseerd op het eerste symbool van een vingerafdruk. Zo wordt elke vingerafdruk die begint met een “0” opgeslagen in een eigen partitie, de vingerafdrukken die beginnen met een “1” in een andere eigen partitie en zo door tot en met de “f”.

Om een partitie te maken moet de tabel worden aangepast met het ALTER commando (MariaDB, 2018). Omdat de partitie verdeeld wordt op basis van de vingerafdruk kan niet het RANGE type gebruikt worden, want deze ondersteunt alleen INTEGER velden en het fingerprint veld is een BINARY veld. Wel kan het RANGE COLUMN partitie type gebruikt worden, want dit type ondersteunt meer typen velden. Om de partities toe te voegen aan de database kan de volgende query gebruikt worden:

ALTER TABLE fingerprints  
PARTITION BY RANGE COLUMNS(fingerprint)  
 (PARTITION partition0 VALUES LESS THAN (0x1000000000000000),  
 PARTITION partition1 VALUES LESS THAN (0x2000000000000000),  
 PARTITION partition2 VALUES LESS THAN (0x3000000000000000),   
 ...  
 PARTITION partitionc VALUES LESS THAN (0xd000000000000000),  
 PARTITION partitiond VALUES LESS THAN (0xe000000000000000),  
 PARTITION partitione VALUES LESS THAN (0xf000000000000000),  
 PARTITION partitionf VALUES LESS THAN MAXVALUE);

Na het maken van de partities is nogmaals getest op de tijd die elke functie nodig had. De resultaten zijn hieronder (afbeelding 22) te zien. Zoals te zien is, liggen de zoektijden van de database nu ongeveer 8 keer zo laag als voor het maken van de partities. Om te kijken of dit geen toeval was, is de test nog vijf keer uitgevoerd. De tijden van de databasefuncties lagen in alle gevallen tussen de 8 en 20 seconden. Door het maken van de partities is er dus een tijdsbesparing van ongeveer 60 tot 70 seconden per test.Afbeelding met schermafbeelding

Automatisch gegenereerde beschrijving

Afb 22

Testfragmenten testen

Nu het matchen minder lang duurt kunnen de tests voor de nauwkeurigheid gedraaid worden. Er is opnieuw een test-set gegenereerd van 200 liedjes met een SNR van -3, 0, 3 en 6 dB. Een testfragment moet minimaal 7 seconden aan muziek bevatten om te voorkomen dat de toegevoegde ruis een lagere sterkte heeft dan zou moeten, omdat de stilte het gemiddelde verlaagt. Zo vallen dus bij veel liedjes het begin en het einde af, omdat er vaak een stilte aan het begin en aan het einde van een nummer zit. Hieronder (tabel 5) zijn de resultaten van deze tests te zien.

|  |  |  |
| --- | --- | --- |
| SNR (dB) | Aantal correct gematcht | Percentage correct |
| -3 | 36 | 18% |
| 0 | 113 | 56.5% |
| 3 | 185 | 92.5% |
| 6 | 192 | 96% |

Tabel 5

Zoals te zien in tabel 5 scoort het programma goed bij signaal-ruisverhoudingen boven de 0dB. Echter, bij een SNR lager dan 3dB is er een sterke afname in nauwkeurigheid (grafiek 1). Opvallend is dat de lijn lineair lijkt te lopen tussen de -3 dB en 3 dB en vervolgens na 3 dB nog langzaam omhoog kruipt.

Grafiek 1

Testen op fout-positieven

Om te testen hoeveel fout-positieven er optreden zijn van 100 nummers die niet in de database stonden testfragmenten gemaakt. Omdat er geen eisen gesteld zijn over het aantal fout-positieven bij een bepaalde SNR, is er geen ruis toegevoegd aan deze fragmenten. Deze testfragmenten zijn vervolgens getest. Wanneer er uit de resultaten kwam dat er een betrouwbare match gevonden was, telt dit als een fout-positief. Van de 100 geteste fragmenten trad er bij 1 fragment een fout-positief op. Dit komt neer op een percentage van 1%.

Omdat Python op vrijwel elk platform geïnstalleerd kan worden (Windows, Linux, Mac) (Python Software Foundation, sd), is dit programma erg flexibel wat besturingssysteem betreft. De tests zijn allemaal uitgevoerd op Windows, maar SONIQ kan dus bijvoorbeeld ook worden uitgevoerd op Ubuntu, wat het oorspronkelijke plan was. De enige software-eis is dan wel dat er een MySQL-compatible database beschikbaar moet zijn, maar ook dit is beschikbaar voor Windows, Linux en Mac (MySQL, sd).

Conclusie

Het gemaakte programma, wat de naam SONIQ heeft gekregen, voldoet niet aan alle gestelde eisen. Hieronder staat per eis of SONIQ eraan voldoet of niet:

Lijst van eisen:

* De geluidsfragmenten die gematcht moeten worden mogen maximaal 10 seconden lang zijn.
  + **Voldaan** – SONIQ accepteert fragmenten van maximaal tien seconden lang
* De software moet met een smartphone te gebruiken zijn.
  + **Voldaan** – Door middel van de Android app is SONIQ op een groot aantal smartphones te gebruiken
* De software moet een catalogus aankunnen van minimaal 2000 nummers.
  + **Voldaan** – SONIQ functioneert bij een catalogus met 2000 nummers.
* De gemiddelde zoektijd van een opgenomen fragment mag maximaal twee seconden zijn, waarvan minimaal 90% onder de drie seconden blijft.
  + **Voldaan** – Indien alleen naar de software gekeken wordt die voor dit ontwerpverslag geschreven is, zit SONIQ ruim onder deze marges. Wanneer er gekeken wordt naar de totale wachttijd, dus ook met de wachttijden van MariaDB, wordt er niet aan de eis voldaan, maar omdat deze software van derden is, is besloten deze niet mee te tellen.
* Bij een signaal-ruisverhouding vanaf 3dB en hoger moet er in minimaal 95% van de gevallen een match gevonden worden.
  + **Niet voldaan** – Bij een signaal-ruisverhouding van 3dB haalt SONIQ een nauwkeurigheid van 92,5%. Dit ligt weliswaar in de buurt van de eis, maar is net niet voldoende.
* Bij een signaal-ruisverhouding van 0 dB moet er in minimaal 80% van de gevallen een match gevonden worden.
  + **Niet voldaan** – SONIQ haalt bij een SNR van 0 dB slechts een nauwkeurigheid van 56,5%, wat ver onder de vereiste 80% ligt.
* Bij een signaal-ruisverhouding van -3 dB moet er in minimaal 70% van de gevallen een match gevonden worden.
  + **Niet voldaan** – De gestelde eis van 70% is niet gehaald. Bij een SNR van   
    -3 dB haalde SONIQ een nauwkeurigheid van 18%. Dit ligt ruim 50% onder de gestelde eis.
* Er mag in maximaal 5% van de gevallen een fout-positief optreden.
  + **Voldaan** – Er trad in 1% van de gevallen een fout-positief op

Zoals hierboven te lezen is, voldoet SONIQ aan meer dan de helft van de eisen. Alleen de nauwkeurigheid viel tegen. Het programma was slecht in staat om zwaar “beschadigde” audio fragmenten te herkennen, maar fragmenten met licht achtergrondgeluid werden wel goed herkend. Verder voldoet SONIQ aan de gestelde capaciteit-eis van 2000 nummers en is via de Android app te gebruiken op een groot deel van de smartphones. Ook is het programma robuust wat betreft fout-positieven, want in slechts 1% van de gevallen is er een fout-positief opgetreden.

Evaluatie

Het ontwerpproces is over het algemeen goed verlopen, maar er zijn ook een aantal dingen die verbeterd zouden kunnen worden.

Zo is het programma door de manier van matchen erg gevoelig voor een verandering in afspeelsnelheid. Dit komt omdat het tijdsverschil tussen twee pieken wordt meegenomen in een vingerafdruk en er bij het evalueren van mogelijke matches wordt gekeken of de onderlinge tijd tussen vingerafdrukken overeenkomt met die in het origineel.

Verder is de database een vertragende factor. Een oplossing hiervoor zou kunnen zijn om zelf een eigen datastructuur te maken om de vingerafdrukken in op te slaan. Dit zou echter wel meer tijd kosten dan wanneer bestaande database software gebruikt wordt. Een alternatief zou zijn om snellere hardware te gebruiken, maar hier zitten weer meer kosten aan verbonden.

Daarnaast is het genereren van een testfragment niet helemaal nauwkeurig. Omdat het programma eerst een fragment van 10 seconden uit een origineel haalt en hier de gemiddelde geluidssterkte van berekent, kan het zijn dat een kort stukje muziek dat erg luid is het gemiddelde omhoog trekt, of andersom. Dit zou ervoor kunnen zorgen dat na het toevoegen van de ruis, afgestemd op de gemiddelde geluidssterkte van het origineel, zachtere gedeeltes van het origineel overstemd worden door de harde ruis of dat er juist een te zachte ruis wordt toegevoegd. Hetzelfde geldt voor de ruis, ook hier kan een gedeelte van de ruis de gemiddelde geluidssterkte erg beïnvloeden en zo een te harde of te zachte ruis laten vormen. Een oplossing hiervoor zou zijn om het fragment van 10 seconden op te splitsen in kleine delen en hier telkens gemiddelden van te pakken en daarop de sterkte van de ruis te baseren.

Als laatste is er niet gelet op de veiligheid van het programma. Omdat de server-modus data accepteert en deze zonder grondig te controleren opslaat en vervolgens verwerkt, is het misschien mogelijk voor een persoon om het programma te laten crashen (DoS) of misschien zelfs willekeurige code uit te voeren (RCE) door malafide data te versturen. Omdat dit programma geen commercieel doeleinde heeft is dit niet heel belangrijk, maar wel een noemenswaardig punt.

Nawoord

Tijdens het maken van dit profielwerkstuk heb ik veel geleerd. Zo ben ik aan de slag gegaan met de programmeertaal Python, waar ik voor dit project weinig ervaring mee had. Daarnaast heb ik geleerd over datastructuren bij grote hoeveelheden data en hoe je deze kunt optimaliseren. Ik vond het leuk om deze dingen te leren en in de praktijk toe te passen en ik denk dat deze dingen mij in de toekomst nog van pas zullen komen, dus dat is mooi meegenomen.

Ik wil de makers van de libraries SciPy, matplotlib, Flask, flask-mysql, Numpy, OkHTTP, GSON, DynamicSineWave en LoadingButtonAndroid bedanken, zonder wiens geweldige werk dit project veel meer tijd zou hebben gekost en waarschijnlijk niet mogelijk zou zijn geweest. Verder wil ik mijn vader bedanken voor zijn goede feedback en tenslotte mijn begeleider, Joris Voets.

Verwijzingen

Akamai. (2015, Februari 15). *Performance Matters.* Opgeroepen op Juli 31, 2018, van www.akamai.com: https://www.akamai.com/us/en/multimedia/documents/content/akamai-performance-matters-key-consumer-insights-ebook.pdf

Crowley, C. (sd). *Audio File Size Calculator*. Opgeroepen op September 30, 2018, van colincrawley.com: https://www.colincrawley.com/audio-file-size-calculator/

Google. (2018a, Juni 6). *AudioFormat*. Opgeroepen op November 13, 2018, van Android Developer Documentation: https://developer.android.com/reference/android/media/AudioFormat#ENCODING\_PCM\_16BIT

Google. (2018b, Juni 6). *AudioRecord*. Opgeroepen op November 13, 2018, van Android Developer Documentation: https://developer.android.com/reference/android/media/AudioRecord

Google. (2018c, Juni 6). *MediaRecorder.AudioSource*. Opgeroepen op Oktober 13, 2018, van Android Developer Documentation: https://developer.android.com/reference/android/media/MediaRecorder.AudioSource

Google. (2018d, Augustus 23). *GSON*. Opgeroepen op November 25, 2018, van Github: https://github.com/google/gson

Haitsma, J., & Kalker, T. (2002, Oktober). *A Highly Robust Audio Fingerprinting System.* Opgeroepen op Augustus 8, 2018, van sci-hub.mu: https://sci-hub.mu/10.1076/jnmr.32.2.211.16746#

MariaDB. (2018). *RANGE Partitioning Type*. Opgeroepen op December 4, 2018, van MariaDB: https://mariadb.com/kb/en/library/range-partitioning-type/

MariaDB. (sd). *RANGE COLUMNS and LIST COLUMNS Partitioning Types*. Opgeroepen op December 4, 2018, van MariaDB: https://mariadb.com/kb/en/library/range-columns-and-list-columns-partitioning-types/

matplotlib. (2018, September 21). *matplotlib.pyplot.specgram*. Opgeroepen op Oktober 1, 2018, van matplotlib.org: https://matplotlib.org/api/\_as\_gen/matplotlib.pyplot.specgram.html

MySQL. (sd). *11.2.1 Integer Types (Exact Value) - INTEGER, INT, SMALLINT, TINYINT, MEDIUMINT, BIGINT*. Opgeroepen op Oktober 14, 2018, van MySQL: https://dev.mysql.com/doc/refman/8.0/en/integer-types.html

MySQL. (sd). *Download MySQL Community Server*. Opgeroepen op December 9, 2018, van MySQL: https://dev.mysql.com/downloads/mysql/

Python Software Foundation. (sd). *Download Python*. Opgeroepen op December 12, 2018, van Python.org: https://www.python.org/downloads/

Ronancher, A. (2018). *Flask*. Opgeroepen op Oktober 24, 2018, van flask.pocoo.org: http://flask.pocoo.org/

SciPy. (2018, Mei 5). *scipy.io.wavfile.read*. Opgeroepen op Oktober 1, 2018, van SciPy.org: https://docs.scipy.org/doc/scipy/reference/generated/scipy.io.wavfile.read.html

Square. (sd). *OkHttp*. Opgeroepen op November 25, 2018, van OkHttp: http://square.github.io/okhttp/

Statista. (2018, Augustus). *• Mobile OS market share 2018 | Statista*. Opgeroepen op Augustus 29, 2018, van Statista: https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

Wang, A. L.-C. (2003). *An Industrial-Strength Audio Search Algorithm.* Opgeroepen op Mei 25, 2018, van ee.columbia.edu: http://www.ee.columbia.edu/~dpwe/papers/Wang03-shazam.pdf

Wikipedia. (2018a, Mei 17). *Acoustic Fingerprint - Wikipedia*. Opgeroepen op Mei 25, 2018, van Wikipedia: https://en.wikipedia.org/wiki/Acoustic\_fingerprint

Wikipedia. (2018b, Juni 28). *Signal-to-noise ratio - Wikipedia*. Opgeroepen op Juli 18, 2018, van Wikipedia: https://en.wikipedia.org/wiki/Signal-to-noise\_ratio

Yang, C. (2001). *Music Database Retrieval Based on Spectral Similarity.* Opgeroepen op Juli 19, 2018, van Music Database Retrieval Based on Spectral Similarity: http://ilpubs.stanford.edu:8090/489/1/2001-14.pdf