Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Timotej Fartek

Implementacija vtičnika Vamp za segmentacijo zvočnih posnetkov

DIPLOMSKO DELO

UNIVERZITETNI ŠTUDIJSKI PROGRAM
PRVE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Mentor: izr. prof. dr. Matija Marolt

Ljubljana, 2018

Copyright. Rezultati diplomske naloge so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavo in koriščenje rezultatov diplomske naloge je potrebno pisno privoljenje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Posnetki, ki jih v radijskih in podobnih programih posnamejo, navadno vsebujejo mešanico govora, petja in instrumentalne glasbe. V okviru diplomske naloge se boste posvetili implementaciji sistema za segmentacijo tovrstnih posnetkov (deljenju na enote, kot so govor, solo petje in tako naprej) kot vtičnik za orodje Sonic Visualiser.

Zahvaljujem se mentorju, izr. prof. dr. Matiji Maroltu, za vse nasvete, ideje in strokovno pomoč, prav tako pa se zahvaljujem družini in vsem, ki so mi tekom študija stali ob strani.

Kazalo

Povzetek

Abstract

1 Uvod **1**

1.1 Pregled področja . . . 2

1.2 Pregled pogosto uporabljenih orodij za analizo zvoka . . . 5

1.3 Motivacija . . . 10

2 Algoritem **11**

2.1 Opis algoritma . . . 11

2.2 Značilke . . . 14

3 Vamp **17**

3.1 Kaj so vtičniki Vamp . . . 17

3.2 Struktura . . . 17

3.3 Vampy . . . 19

3.4 Primer . . . 19

4 Implementacija **23**

4.1 Uporabljene tehnologije . . . 23

4.2 Priprava ogrodja . . . 24

4.3 Branje vhodnih podatkov . . . 26

4.4 Računanje značilk . . . 27

4.5 Statistična analiza . . . 27

4.6 Klasifikacija in segmentacija 28
4.7 Končni izdelek 30
4.8 Performančna analiza 33

5 Sklepne ugotovitve **35**
5.1 Možne izboljšave 35

Literatura **42**

Seznam uporabljenih kratic

kratica	angleško	slovensko
DFT	Discrete Fourier transform	Diskretna Fourierjeva transformacija
FFT	Fast Fourier transform	Hitra Fourierjeva transformacija
MFCC	Mel-frequency cepstral coefficients	Mel frekvenčni kepstralni koeficienti
BIC	Bayesian information criterion	Bayesovski informacijski kriterij

Povzetek

Naslov: Implementacija vtičnika Vamp za segmentacijo zvočnih posnetkov

Avtor: Timotej Fartek

Za arhive zvočnih posnetkov je zelo pomembna digitalizacija, saj se s tem povečuje trajnost hranjenih podatkov. Pri tem se odpre mnogo poti za njihovo semantično obdelavo. Ta naloga se ukvarja s segmentacijo zvočnih posnetkov, torej s smiselnim ločevanjem med govorom in glasbo v zvočnih posnetkih, kar je lahko koristno na primer za radijske postaje ali spletne glasbene knjižnice, kot sta Spotify in Netflix. Tekom te diplomske naloge je bil razvit delujoč algoritem za segmentacijo zvočnih posnetkov, ki za vhod prejme zvočni signal v frekvenčni domeni (to pomeni, da je transformiran z Diskretno Fourierjevo transformacijo), kot izhod pa vrne seznam značilk z določenim časovnim žigom in verjetnostjo, da je na mestu posameznega časovnega žiga zvok klasificiran v razred *glasba.* Implementiran je v obliki vtičnika Vamp in s pomočjo ovojnega vtičnika Vampy sprogramiran v programskem jeziku Python. Analizirala se je tudi hitrost vtičnika v primerjavi z drugimi, že obstoječimi implementacijami segmentacijskega algoritma.

Ključne besede: digitalno procesiranje zvoka, digitalno procesiranje signalov, Vamp, Vampy, segmentacija, Sonic Visualiser

Abstract

Title: Implementation of a Vamp plugin for segmentation of audio recordings

Author: Timotej Fartek

Digitalization is very important for audio data archives as it increases the lifespan and persistence of stored data. In the process multiple options for semantic analysis emerge. This thesis is about segmentation of audio data, specifically the separation between speech and music in audio files which can be useful for instance for radio stations or streaming services such as Spotify and Netflix. Within the scope of this thesis a working segmentation algorithm, which takes a frequency-domain (meaning it is transformed using a discrete fourier transform) input and returns a list of features with their appropriate time stamps and probablities that the input signal at that specific time belongs to the class *music*, was developed. It is implemented as a Vamp plugin and with the help of Vampy, a wrapper plugin, it is programmed in Python. Performance of the developed plugin was also analysed and compared to other pre-existing implementations in Matlab and C#.

Keywords: digital audio processing, digital signal processing, Vamp, Vampy, segmentation, Sonic Visualiser

Poglavje 1

Uvod

V zadnjih letih je postal zelo pomemben vir naših multimedijskih vsebin svetovni splet [19]. Od televizijskih vsebin na zahtevo, kot je Netflix, do spletnih glasbenih knjižnic, kot je Spotify, je zvok zmeraj ena od glavnih komponent. Ustvarjalci vsebin si jih zato prizadevajo med drugim tudi vsebinsko analizirati. To lahko dosežejo tako, da zvočne posnetke klasificirajo in segmentirajo glede na vsebino. Segmentacija se doseže tako, da se zvočni signal razdeli na homogene dele, to je dele s podobno vsebino [26].
Segmentacija je lahko [17]:

- **nadzorovana (angl. supervised):** v tem primeru se uporabi model, ki je bil naučen na množici podatkov z znanimi razredi. Ta način je uporabljen tudi v tej diplomski nalogi.

- **nenadzorovana (angl. unsupervised):** v primeru, ko ni dostopa do učne množice, se segmenti določijo na podlagi gručenja (clustering).

S pravilno segmentiranimi zvočnimi podatki lahko ponudniki spletnih multimedijskih storitev lažje priporočajo izdelke uporabnikom, radijske postaje pa analizirajo terenske posnetke. Tudi arhivi zvočnih posnetkov se širijo zaradi zmeraj večjega števila multimedijskih vsebin in vedno cenejših možnosti hrambe podatkov. Obstaja pa tudi potreba po učinkovitem iskanju, indeksiranju in dostopanju do teh posnetkov [28], [26].

1.1 Pregled področja

Domena analize zvoka je zaradi svoje pomembnosti pogosto deležna pozornosti s strani raziskovalcev. Že več kot 15 let poteka letna konferenca *ISMIR* [5], ki je namenjena pridobivanju podatkov iz glasbe. Na Githubu so zelo popularne knjižnice, kot so AudioKit [3] za jezik Swift ter pyAudioAnalysis [8] in librosa [7] za jezik Python, ki imajo skupno že več kot 6500 zvezdic.

V oddelku za računalništvo univerze *University of Engineering and Technology Taxila* v Pakistanu so razvili algoritem za klasifikacijo in segmentacijo zvoka, ki za svoje delovanje potrebuje zelo majhno učno množico [26]. Predstavljen algoritem ima majhno stopnjo napačne klasifikacije pri majhni množici podatkov, je relativno odporen na šum in primeren za "real-time" uporabo. Algoritem deluje tako, da podatke o zvoku najprej klasificira na *govor* in *ne-govor*, potem pa še dodatno govor na *čisti govor* in *tišino*, ne-govor pa na *glasbo* in *zvok okolja*.

D. Wang, R. Vogt, M. Mason in S. Sridharan so predstavili tehniko segmentacije zvočnih posnetkov glede na identitete govorcev, šume iz ozadja, glasbo in pogoje okolja. To so dosegli s pomočo posplošenega razmerja verjetnosti (angl. generalized likelihood ratio) [28]. Nadgradili so idejo, ki sta jo razvila *Chen* in *Gopalakrishnan* in temelji na Bayesovskem informacijskem kriteriju (angl. Bayesian information criterion – od zdaj naprej BIC). Algoritem so nadgradili tako, da izračuna BIC med dvema sorodnima oknoma, ki sta konstantnih velikosti. Zaradi konstantnih velikosti oken se potem lahko v primeru, da se BIC uporablja kot mera razdalje, ignorira kazen, ki pa jo Chenov algoritem pri hevristični oceni uporablja za kaznovanje kandidatnih modelov glede na njihove kompleksnosti. S to predpostavko potem enačba postane posplošeno razmerje verjetnosti.

Radhakrishnan in *Wenyu Jiang* sta raziskovala način, kako zaznati ponavljajoče se segmente v glasbi s primerjanjem zvočnih prstnih odtisov (angl.

audio fingerprint matching) [25]. Z detekcijo prstnih odtisov (največkrat refren pesmi) se končnemu uporabniku spletne glasbene knjižnice pomaga pri raziskovanju pesmi. To sta dosegla tako, da sta vhodne podatke najprej transformirala v zaporedje zvočnih prstnih odtisov, nato pa v vsakem časovnem koraku primerjala kratka zaporedja prstnih odtisov s preostankom pesmi. Na koncu se najboljši zadetek analizira in zazna ponavljajoče segmente v pesmi.

Matija Marolt je opisal način segmentacije in označevanja etnomuzikoloških terenskih posnetkov. Namen algoritma je zanesljiva aproksimacija "ročnega" segmentiranja terenskih posnetkov, torej, algoritem naj bi posnetke segmentiral na podoben način, kot bi jih človek. Metoda, ki jo je za to izbral, vključuje najprej klasificiranje kratkih delov zvočnega posnetka glede na množico (solo petje, govor, zborsko petje, instrumentalna glasba in zvonjenje), potem se glede na spremembe energije in distribucijo razredov izračuna množica kandidatov za meje med segmenti, na koncu se pa posnetek segmentira s pomočjo determinističnega modela [20].

Pri *BBC Research and Development*[1] so razvili večnamenski vtičnik Vamp, čigar namen je delovati kot skupek algoritmov za pridobivanje značilk iz zvočnih posnetkov. Med njimi je tudi implementacija algoritma za segmentacijo. Za razliko od našega vtičnika ta ne uporablja ovojnega vtičnika Vampy, ampak je implementiran na domorodni (angl. native) način v programskem jeziku C++. Algoritem za segmentacijo deluje tako, da najprej izračuna simetričnost distribucije stopnje ničelnega prehoda na zvočnem signalu. Zatem se poiščejo mesta, kjer se ta distribucija drastično spreminja, in se na vsakem od teh mest segment klasificira kot *govor*, če je povprečna vrednost simetričnosti distribucije večja od določenega praga, ali *glasba*, če je manjša. Zatem algoritem združi segmente, in sicer za vsak segment pogleda, če ima isto vrednost kot njegov predhodnik in ju združi v primeru ujemanja [4].

A. Pikrakis, T. Giannakopoulos in S. Theodoridis so v [22] opisali način ločevanja med glasbo in govorom, ki je splošno računsko nepotraten in zelo

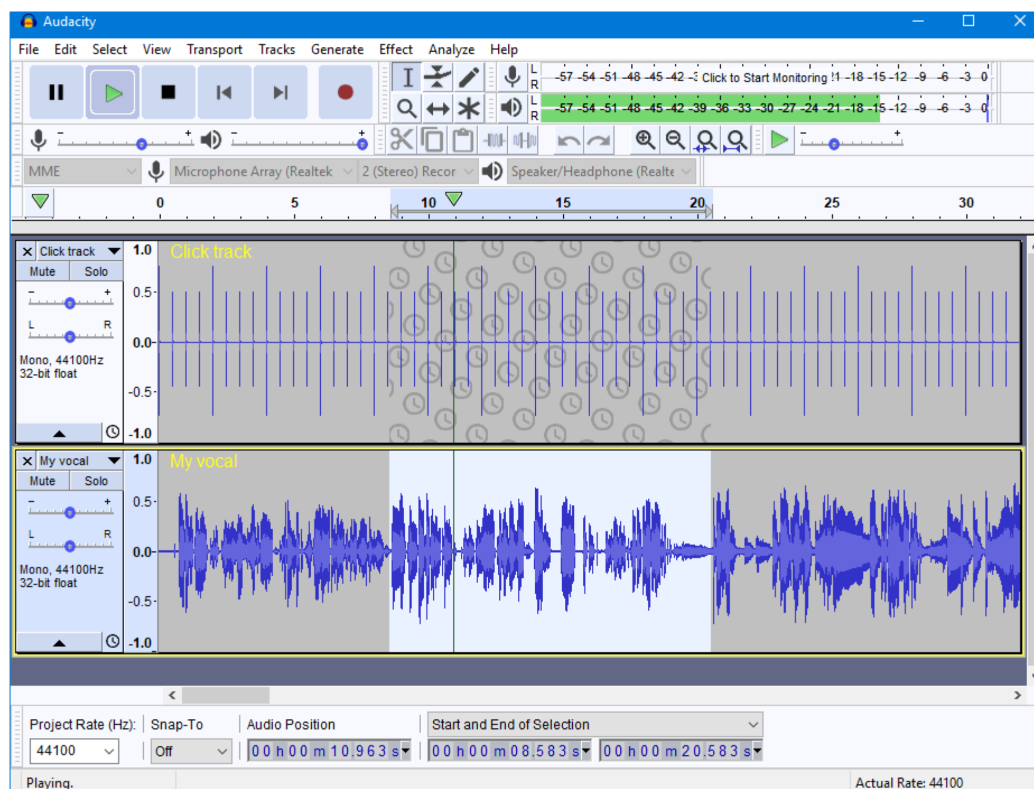
[1] `https://github.com/bbc/bbc-vamp-plugins` [Dostopano: 4. 1. 2018]

natančen. To so dosegli z algoritmom v treh fazah. V prvi fazi se z računsko nepotratnim algoritmom povečevanja regij z veliko točnostjo poiščejo regije signala, kjer nastopata glasba ali govor. Pri tem lahko nekaj regij ostane neklasificiranih. Te regije se nato klasificirajo z bolj računsko zahtevnim algoritmom, ki deluje na principih dinamičnega programiranja. Dobljene regije se v drugi fazi segmentirajo z modelom, ki maksimizira verjetnosti razredov pri podanih značilkah posamičnih okvirjev in omejitvah dolžine segmentov. Pri tem se uporablja Bayesovska mreža za estimacijo pogojne verjetnosti razreda pri podanih značilkah. V tretji fazi se uporabi algoritem za popravljanje napak pri postavljenih mejah med segmenti. S tem načinom segmentacije je bila izmerjena natančnost sistema približno 96 %.

1.2 Pregled pogosto uporabljenih orodij za analizo zvoka

1.2.1 Audacity

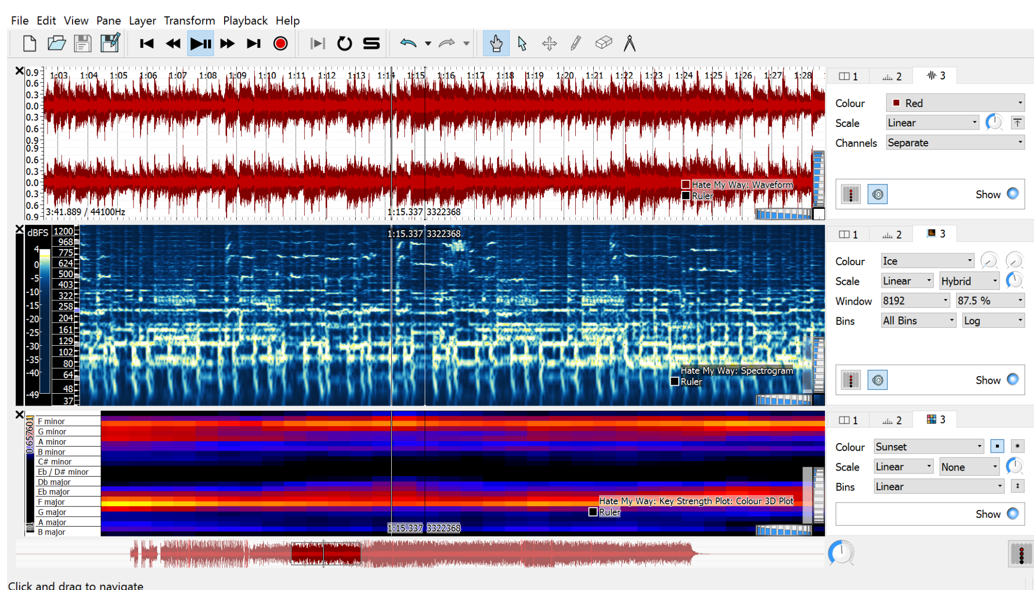
Audacity je orodje za delo z zvočnimi posnetki. Glavni namen tega orodja je snemanje in obdelava zvoka. Vključuje velik nabor orodij za transformacijo zvoka, prav tako pa podpira najpopularnejše vtičnike za transformacijo zvoka. Vključuje tudi nekaj orodij za analizo zvoka in podpira priključitev vtičnikov Vamp.



Slika 1.1: Zajeta slika prikazuje izgled programa Audacity na operacijskem sistemu Windows. [1]

1.2.2 Sonic Visualiser

Sonic Visualiser je odprtokodno orodje, razvito na *Queen Mary University of London*. Njegov cilj je postati prvi program, na katerega naj bi pomislili, ko bi želeli analizirati nek zvočni posnetek. Vključuje velik nabor orodij za analizo zvoka, lahko se ga pa še razširi s pomočjo vtičnikov Vamp, katerih namen pa je prav tako izluščiti vizualno predstavljive značilke iz zvočnih posnetkov.



Slika 1.2: Zajeta slika prikazuje izgled programa Sonic Visualizer na operacijskem sistemu Windows. [9]

Primerjava z Audacity

Aplikaciji Sonic Visualiser in Audacity služita popolnoma drugim namenom. Audacity je orodje, ki je namenjeno splošni obdelavi zvoka, torej transformiranju samega zvočnega signala, medtem ko je namen Sonic Visualiserja pridobivanje podatkov o zvoku z namenom, da se zvočni signal analizira in ne transformira.

1.2.3 Vtičniki

Velikokrat aplikacije za obdelavo in analizo zvoka omogočajo razširitve svojih funkcionalnosti. Te razširitve so po navadi implementirane v obliki vtičnikov, ki jih uporabnik namesti na svoj sistem in s tem doda aplikaciji, kot je na primer Sonic Visualiser, neko dodatno funkcionalnost.

Vtičniki Vamp

So vtičniki za analizo zvoka, ki iz zvočnega signala pridobivajo deskriptivne podatke [11]. Najbolj so znani po tem, da so uporabljeni kot razširitve za program Sonic Visualiser, lahko jih pa uporabljajo tudi druge tako imenovane gostiteljske aplikacije, kot je na primer Audacity. Uporabljajo se specifično za analizo zvočnih podatkov in samega vhodnega signala ne transformirajo. Izhodne podatke lahko podajajo v realnem času (angl. real-time) ali pa na koncu, po končanem celotnem procesiranju. Bolj podrobno so opisani v 3. poglavju.

Vtičniki VST

Virtual Studio Technology (krajše VST) je standard za vtičnike in navidezne glasbene instrumente, razvit pri Steinbergu leta 1996. Namen VST je, da se programsko simulira strojna oprema za transformacijo zvoka, ki se tradicionalno najde v snemalnih studiih. Vtičniki VST delujejo tako, da od gostiteljske aplikacije po kosih prejemajo zvočni signal, ga na poljuben način transformirajo in gostiteljski aplikaciji v realnem času, vračajo rezultat v obliki transformiranega zvočnega signala. Rezultat za uporabnika vtičnika izgleda kot nek zvočni efekt ali zvok navideznega instrumenta [14], [13], [12].

Razlike med vtičniki Vamp, VST in drugimi

Glavna razlika med vtičniki Vamp in ostalimi je v namenu uporabe. Vtičniki Vamp so namenjeni analizi zvoka, ostale popularne tehnologije vtičnikov, kot je VST, pa so namenjene transformaciji zvočnega signala in generiranju

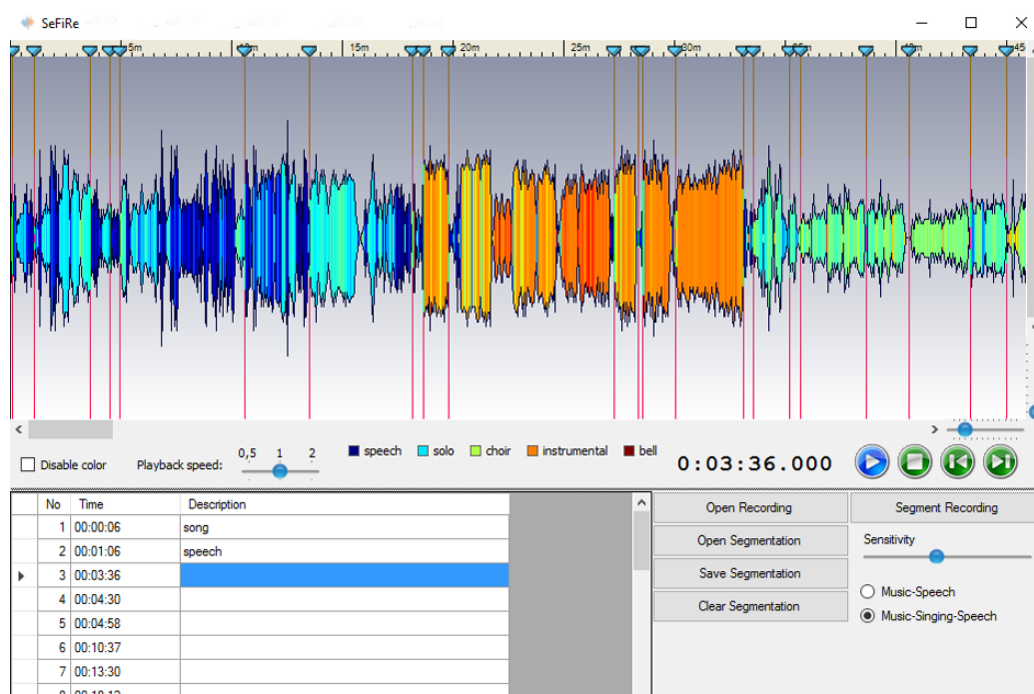
zvočnih efektov. To pomeni, da generirajo nek dodaten zvok, medtem ko vtičniki Vamp generirajo simbolične podatke o vhodnem signalu [11].

Tehnologija vtičnika	Licenca	Namen	Razvijalec	Podprte platforme
Vamp	Tipa BSD	Analiza zvoka	vamp-plugins.org	macOS, Linux, Windows
VST	Lastniška	Transformacija in sinteza zvoka	Steinberg	macOS, Windows
Vtičnik DirectX	Lastniška	Transformacija in sinteza zvoka	Microsoft	Windows
Audio Units	Lastniška	Transformacija in sinteza zvoka	Apple	macOS, iOS, tvOS
Real Time AudioSuite	Lastniška	Transformacija in sinteza zvoka	Avid	macOS, Windows
LV2	ISC	Transformacija in sinteza zvoka	lv2plug.in	macOS, Linux, Windows

Tabela 1.1: Tabela prikazuje razlike med najbolj popularnimi tehnologijami vtičnikov namenjenim obdelavi zvoka [2].

1.2.4 SeFiRe

SeFiRe je orodje, ki je namenjeno segmentaciji terenskih in drugih posnetkov, razvito na *Fakulteti za računalništvo in informatiko v Ljubljani*. Je samostojna aplikacija in vključuje vso funkcionalnost, ki je bila razvita tekom te diplomske naloge v obliki vtičnika Vamp.



Slika 1.3: Zajeta slika prikazuje izgled programa SeFiRe na operacijskem sistemu Windows. [6]

1.3 Motivacija

Potreba po natančnih orodjih za analizo zvoka je zmeraj večja. Izhaja iz tega, da podjetja, kot sta Spotify in Netflix, težijo k personalizaciji multimedijskih vsebin, to je, želijo predvideti, katerih vsebin si posamezni uporabnik trenutno želi. Za to morajo zelo podrobno vsebinsko analizirati vsebine, ki jih nudijo [18]. Radijske postaje imajo podobne težave, s tem da je tam poudarek na indeksiranju podatkov in hitrem iskanju po bazah. Eden najbolj osnovnih korakov za analizo zvoka (na primer procesiranje naravnega jezika) je segmentacija, ki je navadno del predprocesiranja.

Glavni cilj te diplomske naloge je, da se v čim bolj dostopni obliki implementira algoritem za segmentacijo zvočnih posnetkov, ki sicer že obstaja kot samostojni program SeFiRe [20], [6]. Trenutno segmentacijskih orodij, ki bi bila široko dostopna in bi dobro delovala na terenskih posnetkih, še ni. Ravno s tem namenom je naš segmentacijski algoritem implementiran v obliki vtičnika Vamp. Vtičnike Vamp lahko uporabljajo različna orodja oziroma tako imenovane gostiteljske aplikacije (na primer Sonic Visualiser), prav tako pa za uporabo tega vtičnika ni potrebno plačevati nobenih licenc.

Poglavje 2

Algoritem

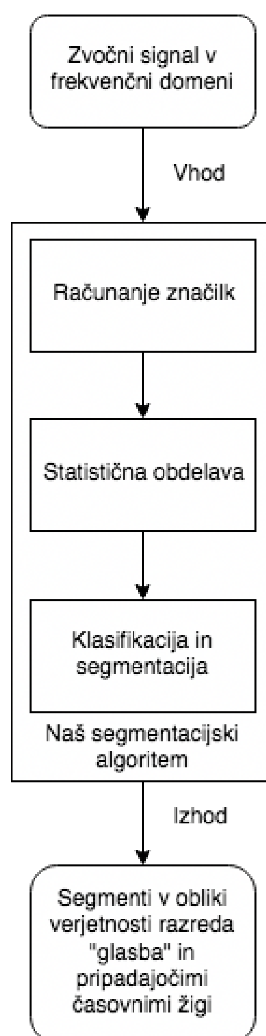
2.1 Opis algoritma

Segmentacijski algoritem je sestavljen iz več medsebojno povezanih delov in poteka v več fazah. Za vhod prejme zvočni signal v frekvenčni domeni po blokih. Zvočni signal v frekvenčni domeni pomeni, da mora biti transformiran s pomočjo diskretne Fourierjeve transformacije, in po blokih pomeni, da se signal podaja algoritmu po kosih in ne v celoti.

Najprej se iz vhodnih podatkov računajo vrednosti značilk. Značilke so si med sabo dovolj različne, da za izračun potrebujejo drugačne pogoje. Značilke *gostota višine zvoka, spektralna entropija, dve vrsti tonalitet, energija in MFCC* za izračun potrebujejo le majhen kos signala, *4 Hz modulacija in tudi tišina* pa celoten signal. Kako ločimo med tema dvema načinoma računanja, je podrobneje razloženo v poglavju o implementaciji algoritma. Poleg samih značilk se izračunajo tudi njihove delte, to je spremembe njihovih vrednosti v oknu (v našem primeru je okno velikosti 5). Tudi za izračun delt značilk potrebujemo celoten signal. Za vsako dano točko na zvočnem signalu se s pomočjo energije določi tudi, če se signal v tistem trenutku lahko smatra kot *tišina*. Po izračunu vseh značilk se naredi statistična analiza s 3-sekundnim oknom. Različne značilke opisujejo različne lastnosti zvoka, zato jih tudi drugače statistično obdelujemo.

Po računanju statistik značilk se morata izvesti še klasifikacija in segmentacija. S pomočjo logističnega klasifikatorja dobimo verjetnosti P_{glasba} na zvočnem signalu. Kot segment se smatra vsak del zvočnega signala, ki je klasificiran, in njegove meje sežejo od svojega časovnega žiga do naslednjega klasificiranega. Trenutna implementacija algoritma ne vsebuje algoritma, ki bi združeval in poenotil segmente glede na podobnost.

Izhod je podan v obliki časovnih žigov in pripadajočih vrednosti. Te vrednosti opisujejo verjetnost, da spada zvočni signal v opazovanem delu v razred *glasba*.



Slika 2.1: Diagram prikazuje korake izvajanja našega segmentacijskega algoritma.

2.2 Značilke

2.2.1 Gostota višine zvoka (angl. pitch density)

Ta značilka opisuje, kako "glasbene" so lastnosti zvoka. Pri računanju se uporabljajo vrednosti realnega dela kepstra, zato je nekoliko odpornejša na šum in motnje v posnetkih [15]. Pri višji vrednosti je večja verjetnost, da je zvok *glasba*, pri nižji pa, da je *govor* ali podobno.
Izračunamo ga po formuli:

$$\frac{1}{N}\sum_{n=k}^{l}|cepst(n)| \tag{2.1}$$

kjer velja

$$k = \left[\frac{fv}{1000}\right], l = \left[\frac{fv}{90}\right] \tag{2.2}$$

in je N skupno število elementov, *cepst(n)* je n-ti koeficient v seznamu realnih kepstrov, *fv* pa je frekvenca vzorčenja.

2.2.2 Tonaliteta (angl. tonality)

Tonaliteta zvoka je organiziran sistem tonov in akordov. En ton (imenovan tonika) postane centralen glede na ostale. S to značilko ugotavljamo, kako tonalen ali šumeč je zvok. Pri višji vrednosti je bolj tonalen, pri nižji pa bolj šumeč [15]. Za izračun uporabimo največjo vrednost filtriranega seznama realnih kepstrov:

$$max(cepst) \tag{2.3}$$

kjer je *cepst* seznam realnih kepstrov in velja:

$$cepst \in \left\{\left[\frac{fv}{1000}\right], \left[\frac{fv}{90}\right]\right\} \tag{2.4}$$

fv pa je frekvenca vzorčenja.

Modificiran algoritem za tonaliteto

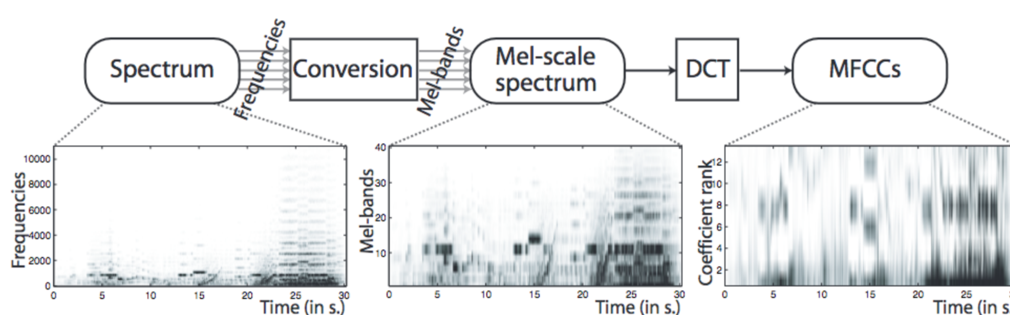
Uporablja se tudi rahlo spremenjena verzija značilke *tonaliteta*. Razlika je v tem, da pri tem algoritmu drugače obtežimo šumeče dele, ki imajo pomotoma visoke kepstralne vrednosti. Med implementacijo bomo značilko poimenovali kar tonaliteta1.

2.2.3 Spektralna entropija (angl. spectral entropy)

Pikrakis je opisal značilko, ki si za delovanje izposoja tehnike iz domene segmentacije slikovnih gradiv [23]. Spektralna entropija oz. tudi *kromatična entropija* se računa na mel-skaliranem signalu. Pri tej značilki nas zanima entropija normalizirane spektralne energije. V primeru *glasbe* je entropija veliko manjša kot pri *govoru*.

2.2.4 MFCC (angl. Mel-frequency cepstral coefficients)

Mel frekvenčni kepstralni koeficienti (krajše MFCC) so transformacija zvočnega signala, ki predstavlja njegovo obliko.



Slika 2.2: Slika prikazuje korake za transformacijo signala v MFCC [21].

So pogosto uporabljene značilke pri razpoznavi govora in pri pridobivanju podatkov iz glasbe [16].

2.2.5 4 Hz modulacija (angl. 4 Hz modulation)

4 Hz modulacija, kot je opisana v [27], [24], je značilka, ki je uporabna pri razpoznavanju govora. Govor ima vrh karakteristične energijske modulacije nekje pri 4 Hz. Za razpoznavo tega iz signala se najprej signal segmentira na kratke okvirje, potem se izračunajo koeficienti Mel spektra (MFCC) in izračuna se energija v 40 zaznavnih kanalih. Energija se potem filtrira, s filtrom centriranim na 4 Hz, sešteje kumulativna vrednost energij vseh kanalov in normalizira glede na povprečno energijo v okvirju. Na koncu se izračuna še modulacija. Izračuna se varianca filtrirane energije v decibelih na 1 sekundi signala. Rezultat je potem modulacijska energija, ki je večja pri govoru kot pri glasbi.

2.2.6 Energija (angl. energy)

S pomočjo izračunane energije ugotavljamo, v katerih delih signala nastopa tišina, to je, ko v signalu ni prisoten noben opazovan zvok, razen šum iz ozadja. Energijo lahko izrazimo kot moč prve komponente diskretne Fourierjeve transformacije zvočnega signala, kar pomeni, da jo lahko izračunamo kot kvadrat absolutne vrednosti prve komponente signala, transformiranega z DFT.

$$E = |F(0)|^2 \tag{2.5}$$

Poglavje 3

Vamp

3.1 Kaj so vtičniki Vamp

Vtičniki Vamp[1] so moduli, ki jih naloži gostiteljska aplikacija, jim poda podatke o zvoku, vtičnik pa nato aplikaciji vrne deskriptivne podatke o podanem vhodu. Vtičniki Vamp tipično vračajo vizualno predstavljive značilke, ki se potem uporabljajo za vizualizacijo, na primer za spektrograme, segmente in tako dalje.

3.2 Struktura

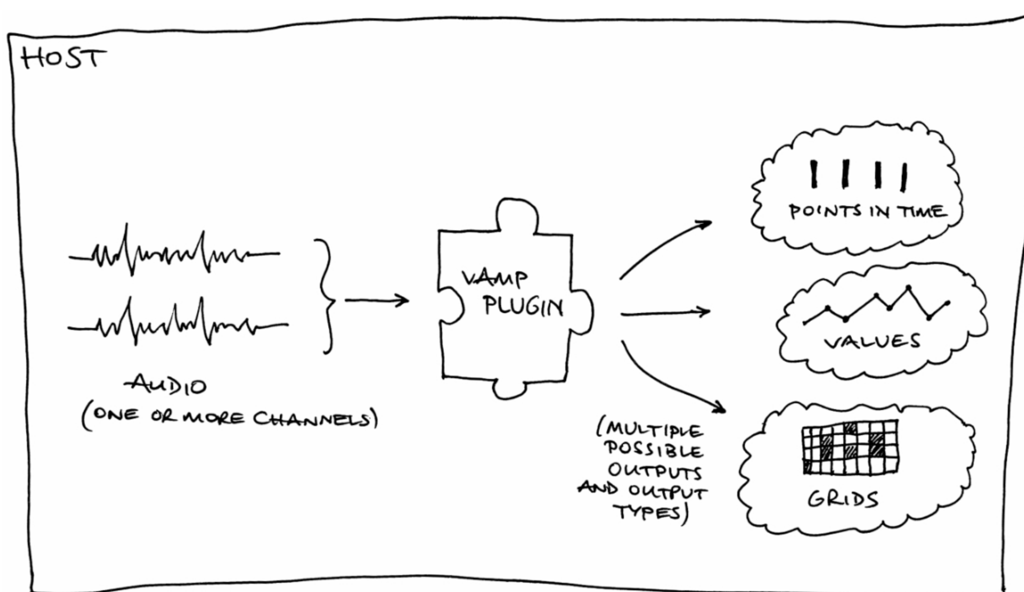
Vsi vtičniki Vamp morajo slediti točno določenim smernicam in imajo zato točno določeno strukturo. Sprogramirani morajo biti v jeziku C++ ali s pomočjo ovojnega vtičnika Vampy v Pythonu.

Morajo vsebovati ustrezne metode za avtorske in splošne podatke (avtor, ime vtičnika, opis, unikaten identifikator, trenutna verzija ...), metode za metapodatke (najmanjše/največje dovoljeno število kanalov, domena vhodnih podatkov, velikost okna, velikost koraka, opis izhodnih podatkov), metodo za nastavitev začetnih parametrov, metodo za ponastavitev in najpomembnejši metodi: *process* (skupaj s pomožno metodo *process2ch* za dvoka-

[1] `http://vamp-plugins.org` [Dostopano: 25. 12. 2017]

nalne vhode) in *getRemainingFeatures*.

Metoda *process* se izvede zmeraj, ko modul dobi nov kos vhodnih podatkov od gostiteljske aplikacije. Namen je, da se v tej metodi sproti računajo razne značilke in podatki, ki se potem ali takoj vračajo v obliki množice ali pa se shranjujejo za konec. Namen metode *getRemainingFeatures* je, da se izračunajo značilke, ki potrebujejo celoto vhodnih podatkov, ali pa se naredi končno čiščenje in filtriranje podatkov. Na koncu se prav tako izvozijo značilke v obliki množice.



Slika 3.1: Slika grafično prikazuje delovanje vtičnikov Vamp [10].

3.3 Vampy

Vampy[2] je ovojni vtičnik, ki omogoča izdelovanje vtičnikov Vamp v jeziku Python. S tem je avtorju vtičnikov omogočeno veliko hitrejše prototipiranje, kot če bi delal z jezikom C++, in dostop do Pythonovih knjižnic, kot je Numpy. Ovojni vtičnik Vampy vzdržuje njegov avtor Gyorgy Fazekas in trdi, da ovojnik vključuje skoraj vso funkcionalnost, ki jo ponuja Vamp API.

3.4 Primer

```
1 import numpy as np
2 import vampy
3 from vampy import *
4
5 class TestPlugin(object):
6     def __init__(self, inputSampleRate):
7         self.vampy_flags = vf_ARRAY | vf_REALTIME
8
9     def initialise(self, channels, step_size,
       block_size):
10        return True
11
12    def reset(self):
13        return
14
15    def getMaker(self):
16        return 'Timotej Fartek'
17
18    def getCopyright(self):
19        return 'Vticnik izdelal Timotej Fartek'
```

[2] http://vamp-plugins.org/vampy.html [Dostopano: 25. 12. 2017]

```
20
21     def getName(self):
22         return 'Test'
23
24     def getPluginVersion(self):
25         return 1
26
27     def getIdentifier(self):
28         return 'test-vticnik-1'
29
30     def getDescription(self):
31         return 'Testni vticnik.'
32
33     def getMinChannelCount(self):
34         return 1
35
36     def getMaxChannelCount(self):
37         return 1
38
39     def getInputDomain(self):
40         return FrequencyDomain
41
42     def getPreferredBlockSize(self):
43         return 1024
44
45     def getPreferredStepSize(self):
46         return 512
47
48     def getOutputDescriptors(self):
49         return OutputList()
50
```

```
51      def getParameterDescriptors(self):
52          return ParameterList()
53
54      def setParameter(self, paramid, newval):
55          return None
56
57      def getParameter(self, paramid):
58          return 0.0
59
60      def process(self, input_buffers, timestamp):
61          return FeatureSet()
62
63      def process2ch(self, input_buffers, timestamp):
64          return FeatureSet()
65
66      def getRemainingFeatures(self):
67          return FeatureSet()
```

Koda 3.1: Koda prikazuje primer strukture vtičnika Vamp, narejenega s pomočjo ovojnega vtičnika Vampy.

Poglavje 4

Implementacija

4.1 Uporabljene tehnologije

Pri izdelavi vtičnika Vamp pri tej diplomski nalogi so uporabljene naslednje tehnologije:

- **Ovojni vtičnik Vampy:** Vampy omogoča programiranje vtičnikov Vamp v programskem jeziku Python. Namen njegove uporabe je hitrejše prototipiranje, omogoča pa tudi uporabo Pythonovih knjižnic, kot je Numpy. Prenese se ga s spletnega mesta Vamp-plugins.org,[1] njegova namestitev pa je preprosta, saj ga je treba samo prenesti v korenski direktorij vtičnikov Vamp na računalniku.

- **Python (z Numpy ter Scipy):** Python[2] je visokonivojski splošnonamenski programski jezik, ki ga je leta 1991 ustvaril Guido van Rossum. Ima preprosto razumljivo sintakso in je zelo primeren tako za hitro prototipiranje kot za natančno analizo in programiranje. Je odprtokodni programski jezik in ima zaradi svoje popularnosti veliko skupnost, ki skrbi za veliko količino vtičnikov in knjižnic. Mednje spada tudi

[1] http://www.vamp-plugins.org/vampy.html
[2] https://www.python.org

knjižnica Numpy,[3] ki spada pod Scipy[4] in vključuje hitre implementacije računsko zahtevnih metod. Scipy vsebuje tudi druge knjižnice za matematične in znanstvene domene.

4.2 Priprava ogrodja

Za primerno delovanje mora imeti Vampy točno določeno strukturo, zato je to tudi prvi korak pri implementaciji algoritma za segmentacijo.

Najprej se lotimo metode *__init__*. V to metodo je od gostiteljske aplikacije podana frekvenca vzorčenja in nastavijo se konstante. V primeru Vampyja se nastavijo tudi zastavice, ki so parametri za ovojni vtičnik Vampy.

```
1 def __init__(self, inputSampleRate):
2     self.vampy_flags = vf_ARRAY | vf_REALTIME
3     self.sample_rate = int(inputSampleRate)
4     self.step_size = 512
5     self.block_size = 1024
6     # itn ...
```

Koda 4.1: Koda prikazuje primer metode __init__.

V drugi vrstici najprej nastavimo Vamp zastavice: *vf_ARRAY* pomeni, da bodo vhodi v metodi *process* in *getRemainingFeatures* podani v obliki Numpy seznama, *vf_REALTIME* pa pomeni, da bodo uporabljeni časovni žigi tipa RealTime. Zatem nastavimo privzete vrednosti, ki jih pa potem tudi lahko prepišemo s podanimi v metodi *initialise*.

V naslednjem koraku nastavimo metodi *initialise* in *reset*. *Initialise* se kliče takoj za *__init__* in v njej se nastavi število kanalov, velikost koraka ter velikost okna, ki ga bo gostiteljska aplikacija uporabljala za transformacijo zvočnega signala v frekvenčno domeno. V tej metodi se lahko inicializirajo tudi katere druge spremenljivke, na primer tiste, ki so odvisne od parametrov

[3] http://www.numpy.org
[4] https://www.scipy.org

te metode. Na koncu metoda vrne *True* ali *False*, odvisno od tega, ali je inicializacija uspešna. V primeru neuspešnosti gostiteljska aplikacija vtičnika ne uporabi. Reset se kliče ob vsaki ponovni uporabi vtičnika za ponastavitev atributov.

```
1 def initialise(self, channels, step_size, block_size):
2         self.step_size = step_size
3         self.block_size = block_size
4         self.hamming_sum = sum(np.hamming(block_size))
5         self.channels = channels
6         self.reset()
7 
8         return True
9 
10 def reset(self):
11         self.initialize_entropy()
12         self.calculate_mel_bank(self.block_size)
13         # itn...
```

Koda 4.2: Koda prikazuje primer metod initialise in reset.

Metodo *reset* pokličemo že takoj ob inicializaciji in se s tem izognemo dupliciranju kode za inicializacijo filtrov za računanje spektralne entropije, MFCC in ostalih.

Zatem nastavimo metode s podatki o avtorju in opisu vtičnika: *getMaker*, *getCopyright*, *getName*, *getPluginVersion*, *getIdentifier*, *getDescription*, zatem *getMinChannelCount* in *getMaxChannelCount*, ki v našem primeru obe vračata 1; *getInputDomain*, ki vrne FrequencyDomain; nato *getPreferredBlockSize*, ki bo pri nas 1024, in *getPreferredStepSize* 512, *getParameterDescriptors* naj vrača prazen seznam ParameterList(); *setParameter* vrne None; *getParameter* 0.0, ker ta vtičnik nima nastavljivih parametrov; in pa *getOutputDescriptors*, kjer se opišejo izhodi iz vtičnika. Podajo se v obliki seznama OutputList.

4.3 Branje vhodnih podatkov

Vamp specifikacija trdi, da lahko vtičnik Vamp pričakuje vhodne podatke v časovni ali v frekvenčni domeni. V primeru časovne domene je to neobdelan zvočni signal, ki ga potem vtičnik lahko poljubno transformira v *process* ali *getRemainingFeatures* metodah. V primeru frekvenčne domene pa je kot vhod v vtičnik podan že transformiran signal s strani gostiteljske aplikacije. V primeru našega vtičnika je zvočni signal transformiran s pomočjo FFT z oknom velikosti 1024 in korakom velikosti 512, šele nato pa je podan kot vhod v metodo *process*. Ta signal potem še dodatno skaliramo s pomočjo vsote Hammingovega okna.

```
1 def initialise(self, channels, step_size, block_size):
2         # ...
3         self.hamming_sum = sum(np.hamming(block_size))
4 
5 def process(self, input_buffers, timestamp):
6         buffered_data = input_buffers[0]
7         scaled_buffer = buffered_data / self.
    hamming_sum * 2
8         fft_mag = np.abs(scaled_buffer)
9         fft_pow = fft_mag**2
10        # ...
```

Koda 4.3: Koda prikazuje primer pridobivanja vhodnih podatkov.

Vsoto Hammingovega okna izračunamo že v metodi *initialise* in se s tem izognemo ponovnemu računanju za vsak kos vhodnega signala. V metodi *process* vzamemo samo prvi zvočni kanal, ker smo specificirali, da naj bo vhod v vtičnik enokanalni, in ga skaliramo. Izračunamo tudi magnitudo in moč zvočnega signala.

4.4 Računanje značilk

Že ob inicializaciji se vnaprej naračunajo tudi filtri, ki jih potrebujemo pri računanju spektralne entropije ter MFCC, in inicializirajo se parametri, potrebni za 4 Hz modulacijo. Večino značilk se v metodi *process* za vsak kos vhodnih podatkov računa sproti. Izjeme so 4 Hz modulacija, tišina in delte značilk. Za izračun teh potrebujemo kompleten nabor vhodnih podatkov, zato se jih računa šele v metodi *getRemainingFeatures*.

Vsaka značilka se računa v svoji funkciji. V metodi *process* se za vsak del vhodnega signala najprej izračunajo značilke *gostota višine zvoka, spektralna entropija, tonaliteta, tonaliteta1, energija in MFCC (vedno, ko omenjamo komponente MFCC, imamo v mislih, da ničto komponento ignoriramo, upoštevamo pa naslednjih 10 komponent)*, v ovojnik za računanje 4 Hz modulacije *self.four_hz_mod_wrapper* dodamo trenutno izračunane koeficiente MFCC, nato pa izračunane značilke dodamo v posamične sezname *self.features['ime_znacilke']*. Na koncu metode *process* vrnemo prazno množico *FeatureSet()*.

Naslednja je metoda *getRemainingFeatures*. Najprej se kliče funkcija *calculate* na ovojniku *self.four_hz_mod_wrapper*, da se izračuna 4 Hz modulacija. Ta se potem shrani v *self.features['four_hz']*, podobno kot ostale značilke. Zatem se računajo delte značilk. Najprej se ustvari filter z velikostjo okna 5, nato pa se za vsako značilko posebej izračunajo delte kot linearni trend.

Naslednja stvar, ki se izračuna, ni značilka zvoka, je pa izpeljan parameter, ki je zelo pomemben – *tišina*. Tišina je zelo vplivna, ker če je nek del signala označen kot *tišina*, ta del potem izključimo iz statistične analize, prav tako pa se ta del zvočnega signala izpusti pri segmentaciji.

4.5 Statistična analiza

V metodi *getRemainingFeatures* se takoj po izračunani tišini kliče funkcija *calculate_statistics*, ki kot rezultat vrne dvoterko *(statistics, skip)*. Značilke se statistično obdelajo v 3-sekundnem oknu in za različne značilke se izračunajo

različne statistike. Pri *MFCC* nas zanimajo povprečne vrednosti prvih štirih komponent, varianca tretje komponente, standardni odklon delt prve in pete komponente ter meana absolutnih vrednosti delt tretje komponente. Pri *tonaliteti* nas zanimata povprečna vrednost in kvocient med varianco ter meano kvadratov vrednosti. Izračunamo tudi povprečno absolutno vrednost delt tonalitete. V sklopu *tonalitete1* izračunamo meano in standardni odklon, prav tako pri *spektralni entropiji*, izračunamo pa tudi varianco delt spektralne entropije. Zanima nas tudi povprečna vrednost *gostote višine zvoka*. Za *4 Hz modulacijo* izračunamo povprečno vrednost ter varianco njenih delt, na koncu pa še standardni odklon delt *energije*. Izračunane statistike se hranijo v seznamih. Funkcija *calculate_statistics* iterira čez celoten zvočni signal in izračuna želene statistike, prav tako pa vsako točko v signalu označi, če se jo naj preskoči pri grajenju končne množice rezultatov.

4.6 Klasifikacija in segmentacija

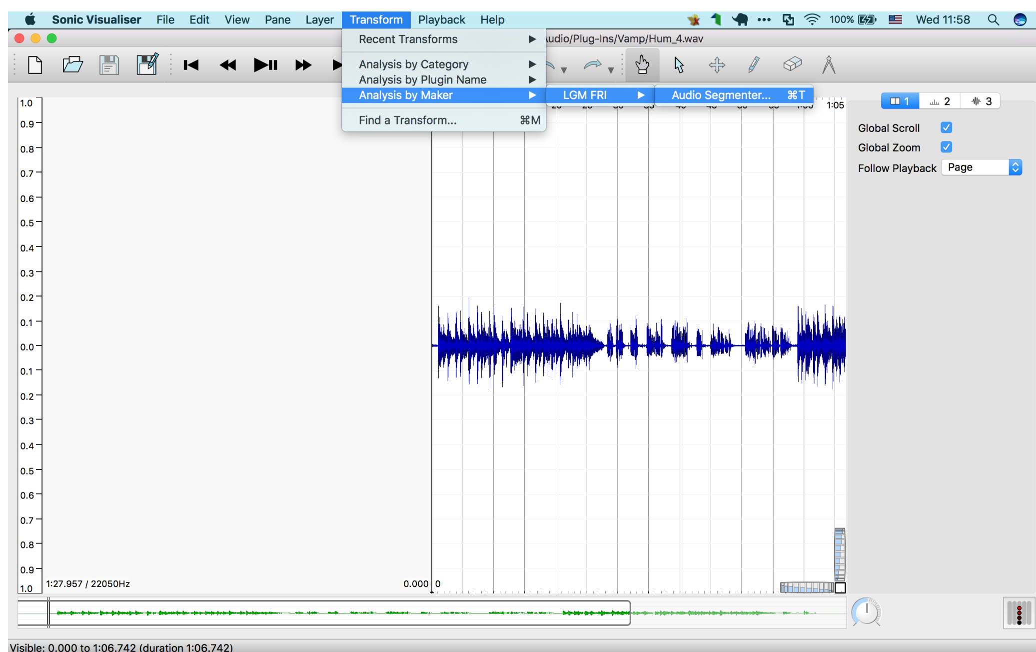
Klasifikacija se izvaja po tem, ko se konča statistična analiza značilk. Funkcija *classify* vsebuje že vnaprej naučene koeficiente, s pomočjo katerih se na logističnem klasifikatorju dobijo ocene. Zatem iteriramo čez dobljene ocene in sestavimo pare verjetnosti v obliki seznama dvoterk *[(verjetnost_glasba, verjetnost_govor), (verjetnost_glasba, verjetnost_govor) . . .]* ter jih vrnemo.

Potem izračunamo velikost koraka, kjer se bodo na signalu postavljale značilke *Feature()*. Inicializiramo izhodno množico *FeatureSet()* in seznam značilk *FeatureList()*, ki je del izhodne množice. Na koncu samo še iteriramo čez celoten zvočni signal in če je v trenutni iteraciji signal označen kot tišina, gremo v naslednjo iteracijo, drugače pa ustvarimo novo značilko *Feature()*, ji pripišemo pripadajočo verjetnost, da spada signal v tem delu v razred *glasba*, izračunamo in nastavimo časovni žig, nastavimo še preostale parametre ter jo dodamo v rezultatni seznam značilk. Dolžina segmenta je od časovnega žiga trenutne značilke do časovnega žiga naslednje značilke. Na koncu metode *getRemainingFeatures* vrnemo množico značilk z vsebovanim seznamom

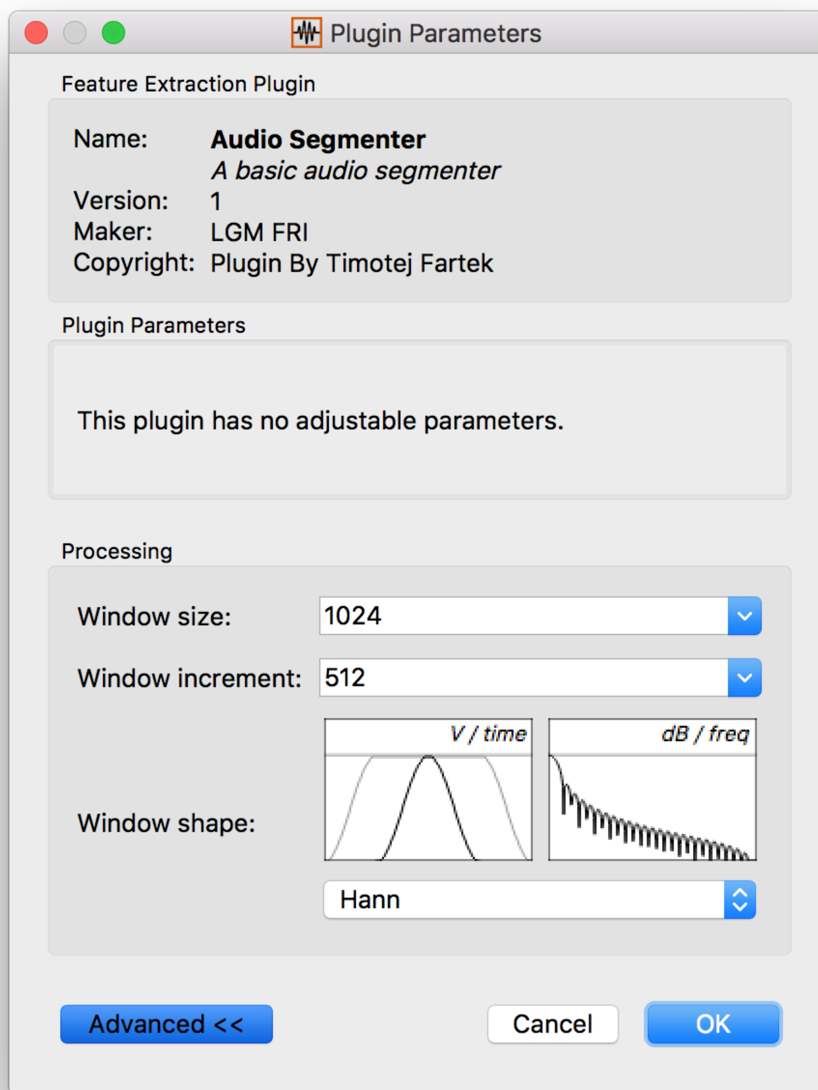
rezultatnih verjetnosti.

4.7 Končni izdelek

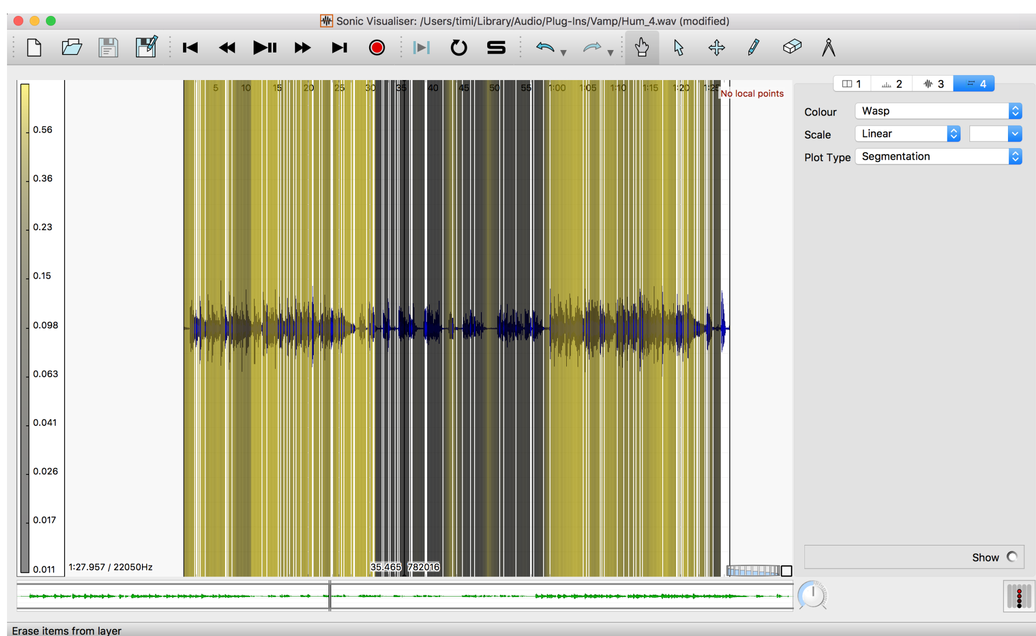
Končni produkt je delujoč vtičnik Vamp (s pomočjo ovojnega vtičnika Vampy), ki izpolnjuje vse teste, ki jih preizkuša Vamp testno orodje *vamp-plugin-tester*.



Slika 4.1: Zajeta slika prikazuje Sonic Visualiser z ustvarjenim vtičnikom.



Slika 4.2: Zajeta slika prikazuje okno z opisom vtičnika. Sonic Visualiser ponuja tudi spreminjanje določenih parametrov, kot so velikost okna, velikost koraka in oblika okna pri DFT, za naš algoritem pa nismo nastavili nobenih specifičnih parametrov, ki bi jih uporabnik lahko spreminjal.



Slika 4.3: Zajeta slika prikazuje primer segmentacije zvočnega posnetka v programu Sonic Visualiser.

4.8 Performančna analiza

Izvedena je bila performančna analiza, kjer se je našo implementacijo v obliki vtičnika Vamp primerjalo z implementacijo v Matlabu in implementacijo v obliki programa SeFiRe.[5]

Analiza je bila izvedena na računalniku s 64-bitnim operacijskim sistemom Windows 10 Pro, procesorjem Intel Core i5 6600K @ 3,50 GHz, spominom 32 GB @ 1200 MHz in diskom Samsung 850 Evo SSD 500 GB.

Vsak algoritem je bil pognan ročno 20-krat za vsako od treh zvočnih datotek. Na koncu se je vzelo povprečno vrednost časa izvajanja. Že pri dvajsetih ponovitvah je bilo očitno, da isti algoritem daje zelo podobne rezultate v zelo podobnem času in da so razlike med implementacijami zelo očitne.

	Signal dolžine 00:09 min.	Signal dolžine 01:28 min.	Signal dolžine 10:00 min.
SeFiRe	0,13 sek.	0,47 sek.	2,84 sek.
Matlab	0,34 sek.	1,25 sek.	4,00 sek.
Sonic Visualiser	1,43 sek.	13,53 sek.	96,50 sek.

Tabela 4.1: Tabela prikazuje povprečne vrednosti časov izvajanja segmentacijskega algoritma v treh različnih implementacijah s tremi vhodi (zvočne datoteke .wav) različnih dolžin. Tukaj je vredno dodati še, da SeFiRe po prvem zagonu za zvočno datoteko zgenerira tudi predpomnilniško datoteko (angl. cache), ki se jo je izbrisalo po vsakem zagonu, zato da se je potem zmeraj znova izvedlo dejansko računanje.

Iz tabele 4.1 je razvidno, da je najhitrejša implementacija v obliki programa SeFiRe, najpočasnejša pa implementacija v obliki Vamp vtičnika. To lahko pripišemo hitrejši naravi prevedenih jezikov C# in C programa SeFiRe, v primerjavi z nekoliko počasnejšim interpretiranim jezikom Matlab in tudi v primerjavi z jezikom Python in ovojnico Vampy našega algoritma.

[5] `http://lgm.fri.uni-lj.si/portfolio-view/sefire/` [Dostopano: 13. 1. 2018]

Naš algoritem niti ni posebej optimiziran, kar se pa lahko doseže s predelavo obstoječih operacij v hitrejše, matrične formule, ki imajo v ozadju implementacije, napisane v hitrejših jezikih, in sicer sta to jezika Fortran in C.

Poglavje 5

Sklepne ugotovitve

Vtičnik daje v glavnem konsistentne rezultate. V primeru, da je gostujoča aplikacija *vamp-simple-host*, vtičnik deluje konsistentno; če pa je gostujoča aplikacija *Sonic Visualiser*, pride včasih do napake in program ne poda vidnih rezultatov. Izvor te napake tekom razvoja ni bil odkrit. Za podan zvočni vhod vtičnik vrne vizualno predstavljive značilke, ki si jih gostiteljska aplikacija lahko interpretira na svoj način.

Vtičnik je bil testiran tudi na konkretnem primeru, in sicer je bil posnet 88-sekundni zvočni posnetek, kjer se prvih 28 sekund igra na kitaro, od 28. do 58. sekunde se narekuje neko besedilo, od 58. do 86. sekunde se spet igra kitara in v zadnjih dveh sekundah se izreče en stavek. Tako v Sonic Visualiserju kot v vamp-simple-hostu kot rezultat dobimo segmente, kjer verjetnost glasbe zelo sovpada z dejanskim zvokom v tistem trenutku, lahko pa pride do manjših odstopanj, ko se izmenjata glasba in govor.

5.1 Možne izboljšave

V razvoju programske opreme je zmeraj prostor za izboljšave. V primeru vtičnika Vamp, ki je bil izdelan tekom te diplomske naloge, bi se lahko nadgradili hitrost in stabilnost vtičnika ter učinkovitost algoritma.

Hitrost procesiranja je lahko zelo pomembna, če se analizira velike zbirke

zvočnih posnetkov. Trenutni vtičnik lahko nadgradimo tako, da optimiziramo trenutne algoritme računanja filtrov in značilk. Trenutni algoritmi v polnosti ne izkoriščajo knjižnic *Numpy* in *Scipy*, ki imata v ozadju zelo hitre implementacije matematičnih operacij. Za še bolj očitno pohitritev lahko celoten vtičnik Vamp prepišemo v jezik *C++* in ne uporabljamo ovojnega vtičnika *Vampy*.

Vtičnik lahko nadgradimo tudi tako, da izboljšamo trenutno implementacijo algoritma za segmentacijo. Trenutno so posamezni končni segmenti zelo majhni, kar pa ni najbolj pregledno. Majhne segmente bi lahko na primer združili v večje glede na prisotnost sosednjih segmentov in tišine. S tem bi se izognili šumu, ki ga imamo pri zelo majhnih segmentih.

Še ena možna izboljšava bi bila nadgradnja stabilnosti vtičnika. Pri testiranju se je izkazalo, da vtičnik ne deluje zmeraj, kot bi moral, ko je gostiteljska aplikacija *Sonic Visualiser*. Včasih se sploh ne prikažejo segmenti ali pa se pokažejo z napačnimi verjetnostmi. V primeru gostiteljske aplikacije *vamp-simple-host*, ki je veliko bolj preprosta, pa so rezultati konsistentni. Težko je ugotoviti točnega krivca, je pa možno, da bi se napako odpravilo, če bi se vtičnik prepisal v jezik *C++* in bi se tako prekinila odvisnost od ovojnega vtičnika *Vampy*.

Slike

1.1 Zajeta slika prikazuje izgled programa Audacity na operacijskem sistemu Windows. [1] 5
1.2 Zajeta slika prikazuje izgled programa Sonic Visualizer na operacijskem sistemu Windows. [9] 6
1.3 Zajeta slika prikazuje izgled programa SeFiRe na operacijskem sistemu Windows. [6] 9

2.1 Diagram prikazuje korake izvajanja našega segmentacijskega algoritma. 13
2.2 Slika prikazuje korake za transformacijo signala v MFCC [21]. 15

3.1 Slika grafično prikazuje delovanje vtičnikov Vamp [10]. 18

4.1 Zajeta slika prikazuje Sonic Visualiser z ustvarjenim vtičnikom. 30
4.2 Zajeta slika prikazuje okno z opisom vtičnika. Sonic Visualiser ponuja tudi spreminjanje določenih parametrov, kot so velikost okna, velikost koraka in oblika okna pri DFT, za naš algoritem pa nismo nastavili nobenih specifičnih parametrov, ki bi jih uporabnik lahko spreminjal. 31
4.3 Zajeta slika prikazuje primer segmentacije zvočnega posnetka v programu Sonic Visualiser. 32

Tabele

1.1 Tabela prikazuje razlike med najbolj popularnimi tehnologijami vtičnikov namenjenim obdelavi zvoka [2]. 8

4.1 Tabela prikazuje povprečne vrednosti časov izvajanja segmentacijskega algoritma v treh različnih implementacijah s tremi vhodi (zvočne datoteke .wav) različnih dolžin. Tukaj je vredno dodati še, da SeFiRe po prvem zagonu za zvočno datoteko zgenerira tudi predpomnilniško datoteko (angl. cache), ki se jo je izbrisalo po vsakem zagonu, zato da se je potem zmeraj znova izvedlo dejansko računanje. 33

Kode

3.1 Koda prikazuje primer strukture vtičnika Vamp, narejenega s pomočjo ovojnega vtičnika Vampy. 19

4.1 Koda prikazuje primer metode __init__. 24

4.2 Koda prikazuje primer metod initialise in reset. 25

4.3 Koda prikazuje primer pridobivanja vhodnih podatkov. 26

Literatura

[1] Audacity. Dosegljivo: http://www.audacityteam.org. [Dostopano: 19. 12. 2017].

[2] Audio plug-in. Dosegljivo: https://en.wikipedia.org/wiki/Audio_plug-in. [Dostopano: 20. 12. 2017].

[3] Audiokit. Dosegljivo: https://github.com/AudioKit/AudioKit. [Dostopano: 19. 11. 2017].

[4] Dokumentacija za "bbc-vamp-plugins". Dosegljivo: https://github.com/bbc/bbc-vamp-plugins/releases/download/v1.1/Documentation.zip. [Dostopano: 20. 12. 2017].

[5] International society for music information retrieval conference (ismir). Dosegljivo: http://ismir.net/conferences.php. [Dostopano: 19. 11. 2017].

[6] Laboratorij za računalniško grafiko in multimedije. Dosegljivo: http://lgm.fri.uni-lj.si. [Dostopano: 19. 12. 2017].

[7] Librosa. Dosegljivo: https://github.com/librosa/librosa. [Dostopano: 19. 11. 2017].

[8] pyaudioanalysis. Dosegljivo: https://github.com/tyiannak/pyAudioAnalysis. [Dostopano: 19. 11. 2017].

[9] Sonic visualiser. Dosegljivo: http://www.sonicvisualiser.org. [Dostopano: 19. 12. 2017].

[10] The vamp audio analysis plugin api: A programmer's guide. Dosegljivo: http://vamp-plugins.org/guide.pdf. [Dostopano: 19. 12. 2017].

[11] Vamp plugins. Dosegljivo: http://vamp-plugins.org. [Dostopano: 20. 12. 2017].

[12] Virtual studio technology. Dosegljivo: https://en.wikipedia.org/wiki/Virtual_Studio_Technology. [Dostopano: 20. 12. 2017].

[13] Vst 3 plug-in sdk. Dosegljivo: https://github.com/steinbergmedia/vst3sdk. [Dostopano: 20. 12. 2017].

[14] Vst3 — steinberg. Dosegljivo: https://www.steinberg.net/en/company/technologies/vst3.html. [Dostopano: 20. 12. 2017].

[15] Z. H. Fu, J. F. Wang, and L. Xie. Noise robust features for speech/music discrimination in real-time telecommunication. In *2009 IEEE International Conference on Multimedia and Expo*, pages 574–577, June 2009.

[16] Todor Ganchev, Nikos Fakotakis, and George Kokkinakis. Comparative evaluation of various mfcc implementations on the speaker verification task. In *in Proc. of the SPECOM-2005*, pages 191–194, 2005.

[17] Theodoros Giannakopoulos. pyaudioanalysis: 5. segmentation. Dosegljivo: https://github.com/tyiannak/pyAudioAnalysis/wiki/5.-Segmentation. [Dostopano: 19. 11. 2017].

[18] Lev Grossman. How computers know what we want — before we do. *TIME*, 2010.

[19] Katie Young Jason Mander. Digital vs. traditional media consumption, Q1 2017.

[20] Matija Marolt. International society for music information retrieval conference - probabilistic segmentation and labeling of ethnomusicological field recordings. 2009.

[21] Petri Toiviainen Olivier Lartillot. A matlab toolbox for musical feature extraction from audio. In *10th Int. Conference on Digital Audio Effects (DAFx-07), Bordeaux, France, September 10-15, 2007.*

[22] A. Pikrakis, T. Giannakopoulos, and S. Theodoridis. A speech/music discriminator of radio recordings based on dynamic programming and bayesian networks. *IEEE Transactions on Multimedia*, 10(5):846–857, Aug 2008.

[23] Aggelos Pikrakis, Theodoros Giannakopoulos, and Sergios Theodoridis. A computationally efficient speech/music discriminator for radio recordings. In *in Proc. of of the 7th International Conference on Music Information Retrieval (ISMIR 06)*, pages 107–110, 2006.

[24] Julien Pinquier, Christine Sénac, and Régine André-obrecht. Robust speech / music classification in audio documents. In *in Proc. ICSLP'02*, 2002.

[25] R. Radhakrishnan and W. Jiang. Repeating segment detection in songs using audio fingerprint matching. In *Proceedings of The 2012 Asia Pacific Signal and Information Processing Association Annual Summit and Conference*, pages 1–5, Dec 2012.

[26] Muhammad Rashid Muhammad Haroon Yousaf Saadia Zahid, Fawad Hussain and Hafiz Adnan Habib. Optimized audio classification and segmentation algorithm by using ensemble methods. *Mathematical Problems in Engineering, vol. 2015*, 2015.

[27] E. Scheirer and M. Slaney. Construction and evaluation of a robust multifeature speech/music discriminator. In *1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 1331–1334 vol.2, Apr 1997.

[28] D. Wang, R. Vogt, M. Mason, and S. Sridharan. Automatic audio segmentation using the generalized likelihood ratio. In *2008 2nd Inter-*

national Conference on Signal Processing and Communication Systems, pages 1–5, Dec 2008.