POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Elektronika (EKA)
SPECJALNOŚĆ: Inżynieria akustyczna (EIA)

PRACA DYPLOMOWA INŻYNIERSKA

Projekt i implementacja aplikacji automatycznej klasyfikacji utworów muzycznych z różnych epok

Design and implementation of the application for process of automatic classification of classical music from different eras

AUTOR:
Patryk Grzybała

PROWADZĄCY PRACĘ:
dr inż. Maciej Walczyński, PWr

OCENA PRACY:

WROCŁAW 2019

Spis treści

Rozdział 1

Wstęp

Celem pracy było stworzenie aplikacji pozwalającej na klasyfikację klasycznych utworów muzycznych z wykorzystaniem algorytmów uczenia maszynowego oraz formatu MusicXML.

W niniejszej pracy inżynierskiej omawiany jest temat automatycznego rozpoznawania utworów muzyki klasycznej z różnych epok wykorzystując do tego algorytmy uczenia maszynowego. Praca omawia wyniki uzyskane podczas tworzenia aplikacji oraz omawia napotkane problemy podczas jej tworzenia.

Uczenie maszynowe zyskuje na coraz większej popularności. Uczenia maszynowe jako dziedzina nauk o sztucznej inteligencji pojawiła się już w latach pięćdziesiątych ubiegłego wieku kiedy Arthur Lee Samuel po raz pierwszy, w roku 1959, użył terminu "uczenie maszynowe". Program Arthura uczył się grać w prostą grę planszową - warcaby. W kolejnych latach dzięki uczeniu maszynowemu odkryto nieznane molekuły związków organicznych, a wyniki tychże badań ukazały się w prasie naukowej i po raz pierwszy nie były to badania wykonane przez człowieka. Obecnie uczenie maszynowe jest wykorzystywane w wielu dziedzinach z ogromnymi sukcesami.

1.1 Zakres pracy

W zakres pracy wchodzą:

- pozyskanie parametrów opisujących utwór muzyczny z plików MusicXML pozwalających na reprezentowanie zachodniej notacji muzycznej,
- stworzenie bazy danych plików muzycznych z podziałem na różne epoki,
- wykorzystanie istniejących algorytmów uczenia maszynowego do jak najlepszej klasyfikacji utworów muzycznych,

- stworzenie interfejsu wiersza poleceń pozwalającego na sklasyfikowanie utworu.

1.2 Zawartość pracy

Rozdział 2 ("Muzyka klasyczna") opisuje pokrótce epoki w muzyce oraz skupia się na zapisie nutowym.

Rozdział 3 ("Baza plików muzycznych MusicXML") mówi o tworzeniu bazy plików muzycznych wykorzystywanych do późniejszej nauki klasyfikatorów uczenia maszynowego oraz o formacie MusicXML.

Rozdział 4 ("Uczenie maszynowe") zawiera podstawowe informacje o uczeniu maszynowym, jego rodzajach, niektórych algorytmu klasyfikujących oraz wstępnym przetwarzaniu danych.

Rozdział 5 ("Implementacja aplikacji") przedstawia środowisko w jakim została napisana aplikacja, omawia w skrócie bibliotekę *music21* oraz opisuje jak została stworzona aplikacja.

Rodział 6 ("Podsumowanie") omawia uzyskane wyniki klasyfikacji, wnioski, które nasunęły się podczas pisania pracy oraz możliwości rozwoju czy usprawnienia aplikacji.

Rozdział 2

Muzyka klasyczna

Muzyka towarzyszyła nam od czasów prehistorycznych, grano podczas ceremonii religijnych oraz do zabawy. Najstarszy wiernie odtworzony utwór muzyczny pochodzi z państwa Hurytów, jest to "Hymn do Nikkal" i datowany jest na rok 1400 p.n.e., natomiast najstarszy kompletny utwór muzyczny, który zachował się do czasów współczesnych datowany jest na XI wiek.

2.1 Epoki w muzyce klasycznej

2.1.1 Średniowiecze

Muzyka w średniowiecznej Europie nie była czymś pospolitym. Muzykę uważano za naukę, a także większość autorów nie podpisywała się pod swoimi dziełami, miała na to wpływ filozofia małości człowieka wobec Boga. W hierarchii społecznej teoretycy muzyczni zajmowali znacznie większą pozycję niż kompozytorzy, ci ostatni byli tylko rzemieślnikami.

W końcówce IX wieku zaczęła pojawiać się notacja muzyczna służąca do zapisu chorału gregoriańskiego zwana notacją neumatyczną. Znaki w tej notacji odpowiadały pojedynczym dźwiękom albo ich grupom. Umieszczano je nad tekstem śpiewanym. Miało to określić przybliżony kształt melodii.

Guido z Arezzo wprowadził do tej notacji dwie równoległe linie i dwie litery C - przy wyższej linii oraz F przy niższej linii. Litery te oznaczały wysokość dźwięku zapisanych na danych liniach. W późniejszych latach, pod koniec XI wieku, dodano dwie, a następnie jeszcze jedną linię. Tak narodziła się znana nam ówcześnie pięciolinia.

2.1.2 Renesans

Za początek renesansu w muzyce uznaje się rok 1400, kiedy zaczęto wykorzystywać nieznane w średniowieczu techniki kompozytorskie. Muzyka tego okresu była przeznaczona głównie dla bogatego mieszczaństwa oraz na dwory władców. Zaczynają się pojawiać utwory świeckie: pieśni miłosne, madrygały i utwory okolicznościowe. Muzyka zmieniła swój status z nauki na sztukę.

Wśród muzyki sakralnej dominowały msze i motety, natomiast w muzyce świeckiej dominował chanson oraz madrygał. Główne instrumenty utożsamiane z renesansem to puzon, lutnia, viola da gamba oraz organy.

Dodatkowym czynnikiem rozwijającym muzykę renesansową było wynalezienie maszyny drukarskiej co pozwoliło na proste kopiowanie nut i rozpowszechnienie muzyki na większą skalę.

2.1.3 Barok

W roku 1600 nastąpił przełom w muzyce, gdy we Florencji wystawiono pierwsze zachowane dzieło operowe "Dramma per musica" Jacopo Periego. Przyczyniło się to do powstania nowego gatunku - opery. Nowa epoka w muzyce miała odróżniać się od poprzedniej przede wszystkim jej ekspresywnością i indywidualnym charakterem. Muzyka baroku miała za zadanie wyrażanie emocji, a także pobudzanie w słuchaczu uczuć zwanych afektami.

Na przełomie XVII i XVIII wieku muzyka stawała się bardziej monumentalna. Utwory były coraz dłuższe, wieloczęściowe, były wykonywane przez większą ilość muzyków. Muzyka instrumentalna bardzo się rozwijała, w tym okresie powstały między innymi sonaty oraz koncerty solowe jako gatunki. Warto również wymienić tu nazwiska takich twórców jak Antonio Vivaldi, Jerzy Fryderyk Händel oraz Jan Sebastian Bach, którzy są ikonami muzyki z tamtego okresu.

2.1.4 Klasycyzm

Za początek klasycyzmu uznaje się śmierć Jana Sebastiana Bacha w roku 1750. Do najbardziej znanych kompozytorów z tego okresu należeli Józef Haydn, Wolfgang Amadeusz Mozart, a także Ludwig van Beethoven oraz Franz Schubert, których twórczość ciężko jednak przypisać jednoznacznie do okresu klasycyzmu, gdyż są pionierami również epoki romantycznej.

W epoce klasycyzmu udoskonalano gatunki i formy, takie jak sonaty, symfonie, koncerty czy kwartety. Jest to epoka bardzo różnorodna posiadająca wiele szkół i stylów.

Początek tej epoki to przede wszystkim styl *galant*, muzyka w tym stylu miała być przyjemna, prosta ale też elegancka, a jej głównym zadaniem było wprowadzenie słuchacza w dobry nastrój. Kompozytorzy rezygnowali z polifonii, która była wszechobecna w stylu barokowym.

2.1.5 Romantyzm

XIX wiek był rozkwitem nurtu romantycznego w Europie, rozpoczynając od nazwiska Ludwiga van Beethovena, w którego muzyka ukształtowała późniejszych twórców romantycznych. XIX był też początkiem twórczości wirtuozowskiej - *brillant*. Głównymi inspiracjami muzyki tego okresu był folklor, egzotyka, natura, filozofia, literatura oraz inne sztuki. Odmienił się status społeczny kompozytora i wykonawcy, a także muzyki w pojmowaniu społeczeństwa. Publiczne koncerty stawały się coraz bardziej popularne.

Romantyczni twórcy starali się tworzyć swoje unikatowe style, odchodzono od szeroko przyjętych schematów form muzycznych i zasad porządkujących przebieg dzieła muzycznego. Gatunki muzyczne nie były tak dystynktywne. Teksty utworów romantycznych często były poświęcone tematom miłości oraz samotności, a inspiracją do nich była twórczość ówczesnych poetów.

2.1.6 XX wiek

Muzyka XX wieku stanowi bunt wobec muzyki z wcześniejszych okresów. Charakteryzuje się poszukiwaniem nowości, inności oraz łamaniem zasad. Poprzednie epoki odznaczały się tzw. *tonalnością funkcyjną* co oznacza, że panowała w nich swego rodzaju hierarchia, zdarzenia muzyczne były skorelowane z tym co działo się wcześniej, a także pozwalały przewidywać co stanie się później. Uznano jednak, należy szukać nowych zasad. Jednak cechą, która łączy muzykę wieku XX z muzyką lat poprzednich były uprawiane formy i gatunki muzyczne takie jak symfonie czy opery. Muzyka tego okresu jest bardzo zróżnicowana.

2.2 Zapis nutowy

W tej pracy pliki wykorzystane do klasyfikacji zawierają między innymi informacje o wysokości dźwięku i wartości rytmiczne nut, są to dwie główne informacje wykorzystane do parametryzacji utworów muzycznych.

2.2.1 Wysokość dźwięku

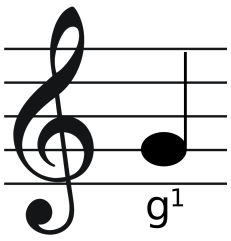
Wysokość dźwięku danej nuty określa jej częstotliwość tonu podstawowego. Na partyturze wysokość dźwięku danej nuty określa jej położenie na pięciolinii, klucz oraz znaki chromatyczne, czyli krzyżyki, bemole i kasowniki.
W tabeli 2.1 przedstawiono ilość drgań dla dźwięków w oktawie razkreślnej.

Tabela 2.1 Dzwięki i ich częstotliwości w oktawie razkreślnej

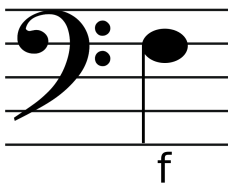
Dźwięk	c^1	d^1	e^1	f^1	g^1	a^1	h^1	c^2
Częstotliwość [Hz]	261,6	293,7	329,6	349,2	391,9	440,0	493,9	523,2

Klucz

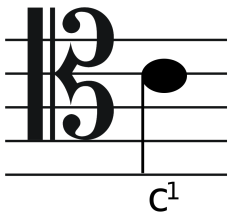
Klucz, umieszczany na początku pięciolinii, przyporządkowuje liniom odpowiednie wysokości dźwięków. Istnieją trzy rodzaje kluczy G, F oraz C, które zostały zaprezentowane odpowiednio na rysunkach 2.1, 2.2 oraz 2.3.



Rysunek 2.1 Klucz G.



Rysunek 2.2 Klucz F.



Rysunek 2.3 Klucz C.

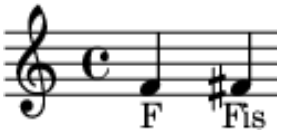
Klucz G wskazuje położenie dźwięku g^1, klucz F wskazuje położenie dźwięku f, a klucz C położenie dźwięku c^1 na pięciolinii. Klucze ułatwiają zapis dźwięków na odpowiednie partie, na przykład nuty dla instrumentów nisko brzmiących takich jak kontrabas zapisuje się z użyciem klucza basowego, a klucza wiolinowego używa się do zapisu nut dla wysoko brzmiących skrzypiec. Inaczej pięciolinia nie wystarczyłaby do zapisu wszystkich dźwięków, nawet z liniami dodatkowymi.

Krzyżyk

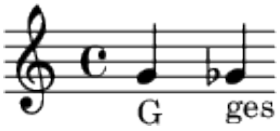
Krzyżyk podnosi dźwięk o pół tonu, czyli sprawia, że dźwięk jest podniesiony o $\sqrt[12]{2}$ raza.

Bemol

Bemol jest przeciwieństwem krzyżyka i obniża dźwięk o pół tonu.



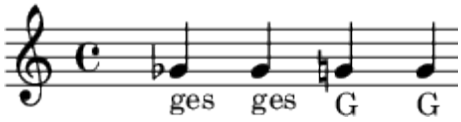
Rysunek 2.4 Symbol krzyżyka przy nucie.



Rysunek 2.5 Symbol bemolu przy nucie.

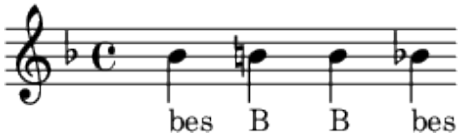
Kasownik

Kasownik, jak sama nazwa wskazuje, kasuje znaczenie wyżej wymienionych znaków chromatycznych. Postawienie np. bemolu przed nutą G oznacza, że wszystkie nuty G w danym takcie będą obniżone o pół tonu staną się nutami ges, z kolei postawienie kasownika przed nutą ges kasuje znaczenie bemolu i dźwięk zostaje podniesiony o pół tonu w górę. Ilustruje to rysunek 2.6.



Rysunek 2.6 Zmiana tonu dzwięku ges przez kasownik.

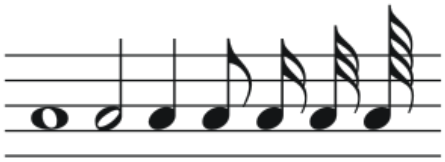
Znaki chromatyczne można jednak stawiać także za kluczem, obowiązują wtedy w całym utworze lub są tymczasowo usuwane przez kasownik.



Rysunek 2.7 Zmiana tonu dźwięku bes przez kasownik.

2.2.2 Wartości nut

Wartości rytmiczne nut określają długość ich trwania. Podstawową jednostką jest *cała nuta*, według której określone są pozostałe nuty. Cała nuta nie ma absolutnie zdefiniowanego czasu trwania, jest on definiowany przez *tempo* utworu. Cała nuta trwa pełny takt w metrum $^2/_2$ czy $^4/_4$. Cała nuta zawiera dwie półnuty, cztery ćwierćnuty, osiem ósemek itd.

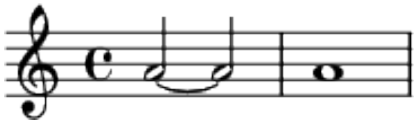


Rysunek 2.8 Symbole nut przedstawiające różne wartości nut, kolejno od lewej: cała nuta, półnuta, ćwierćnuta, ósemka, szesnastka, trzydziestodwójka, sześćdziesięcioczwórka.

Wartości rytmiczne nut mogą być zmieniane. Podobnie jak zwiększana jest wysokość dźwięku przez krzyżyki tak długość nuty może być wydłużona przez łuk lub kropkę.

Łuk

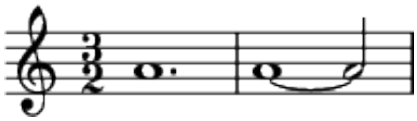
Łuk łączy dwie nuty tylko i wyłącznie o tej samej wysokości, ich wartości się sumują. Łuki łączące nuty o różnych wysokościach mają inne znaczenie. Na rysunku 2.9 zaprezentowano połączenie dwóch półnut w jedną całą nutę. Dźwięki w obu taktach brzmią tak samo.



Rysunek 2.9 Łączenie dwóch półnut do całej nuty przez łuk.

Kropka

Kropka po prawej stronie nuty wydłuża jej wartość o połowę wartości nuty przy której została napisana, to znaczy, że cała nuta z kropką trwa tyle co cała nuta i półnuta, półnuta z kropką trwa tyle co półnuta i ćwierćnuta itd. Zależność ta została zaprezentowana na rysunku 2.10. Kropka pisana nad nutą ma inne znaczenie.



Rysunek 2.10 Wydłużenie wartości całej nuty przez kropkę.

Rozdział 3

Baza plików muzycznych MusicXML

W ramach niniejszej pracy musiała zostać stworzona baza plików muzycznych utworów z różnych epok. Pliki muzyczne zgodnie z założeniami były w formacie MusicXML i zostały pobrane ze stron:

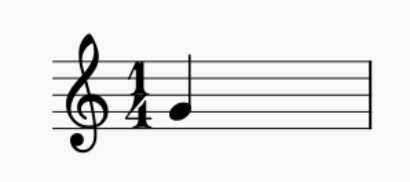
- `https://www.cpdl.org/wiki/`
- `https://musescore.com/sheetmusic`

3.1 Format MusicXML

Format MusicXML to cyfrowy format do zapisu notacji nutowej i łatwego udostępniania zachodnich utworów muzycznych. Celem tego formatu jest stworzenie uniwersalnego nośnika do udostępniania nut. Obecnie wiele programów komputerowych wspiera otwieranie plików MusicXML. Przed MusicXML jedynym powszechnie stosowanym formatem do zapisywania notacji było MIDI, jednak ten format nie wspiera wielu cech notacji nutowej. MusicXML używa języka znaczników XML do zapisu informacji, który został wybrany ze względu na niezależność od platformy i jest świetny do reprezentowania dużej ilości strukturalnych danych co wpasowuje go idealnie do reprezentacji partytury. Dodatkowym atutem tego formatu jest fakt, że nie ma potrzeby tworzenia nowych parserów tego języka, gdyż istnieją one już od dłuższego czasu.

Pliki MucsicXML użyte podczas tworzenia bazy muzyki używają skompresowanego formatu .mxl, który pozwala zaoszczędzić około 20 razy więcej miejsca, przykładowo cztery strony nut w nieskompresowanym formacie .musicxml zajmują nawet 518kB pamięci, a przy użyciu skompresowanego formatu .mxl tylko 19kB co jest nawet mniejszą ilością w porównaniu do reprezentacji MIDI, która wynosi 21kB, a jednocześnie zawiera mniej informacji.

3.1.1 Przykład pliku MusicXML



Rysunek 3.1 Przykład formatu .mxl odtworzonego za pomocą programu MuseScore 3.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE score-partwise PUBLIC
    "-//Recordare//DTD MusicXML 3.1 Partwise//EN"
    "http://www.musicxml.org/dtds/partwise.dtd">
<score-partwise version="3.1">
  <part-list>
    <score-part id="P1">
      <part-name>Music</part-name>
    </score-part>
  </part-list>
  <part id="P1">
    <measure number="1">
      <attributes>
        <divisions>1</divisions>
        <key>
          <fifths>0</fifths>
        </key>
        <time>
          <beats>1</beats>
          <beat-type>4</beat-type>
        </time>
        <clef>
          <sign>G</sign>
          <line>2</line>
        </clef>
      </attributes>
      <note>
        <pitch>
          <step>G</step>
          <octave>4</octave>
```

```
        </pitch>
        <duration>1</duration>
        <type>quarter</type>
      </note>
    </measure>
  </part>
</score-partwise>
```

Na rysunku 3.1 przedstawiono graficzną reprezentacje powyższego kodu. Przeanalizujmy wszystkie części tego kodu.

```
<?xml version="1.0" encoding="UTF-8"?>
```

Wymagana deklaracja dokumentu wszystkich plików XML, z ustalonym kodowaniem znaków na UTF-8 posiadającym zbiór znaków ASCII.

```
<!DOCTYPE score-partwise PUBLIC
    "-//Recordare//DTD MusicXML 3.1 Partwise//EN"
    "http://www.musicxml.org/dtds/partwise.dtd">
```

Te linijki mówią o tym, że używamy MusicXML, jednocześnie specyfikujemy lokalizację DTD(ang. Document Type Definition), czyli dokumentu definiującego strukturę plików XML. Pozwala to na rozpoznanie plików MusicXML przez aplikacje.

```
<score-partwise version="3.1">
```

To jest główny *korzeń* (ang. *root*) dokumentu, element <score-partwise> składa się z części, a każda z części składa się z taktów. Atrybut elementu *version* definiuje, którą wersję MusicXML używamy.

```
<part-list>
  <score-part id="P1">
      part-name>Music</part-name>
  </score-part>
</part-list>
```

Powyższy znacznik <*part-list*> jest nagłówkiem zawierającym liczbę części w partyturze, w powyższym przykładzie zawiera on tylko jeden element, który jako wymagany atrybut zawiera ID części oraz również wymagany element <*part-name*> zawierający nazwę części.

```
<part id="P1">
```

Tu zaczyna się pierwsza i jedyna część w utworze, wymagany atrybut *id* musi odnosić się do ID zdefiniowanym w elemencie nagłówkowym <*part-list*>.

```
<measure number="1">
```

Rozpoczynamy takt numer 1.

```
<attributes>
```

Element <*attributes*> zawiera informacje potrzebne do interpretacji wartości muzycznych nut.

```
<divisions>1</divisions>
```

Każda nuta w MusicXML zawiera swój element określający jej czas trwania. Element <*divisions*> dostarcza jednostkę pomiaru długości elementu ze względu na podział na ćwierć nuty. W tym przykładzie mamy tylko jedną ćwierćnutę o elemencie <*duration*> równym 1. Element <*divisions*> powinien być równy długości ćwierćnuty.

```
<key>
    <fifths>0</fifths>
</key>
```

Element <*key*> reprezentuje znaki przykluczowe, wartość 0 oznacza tonacje C-dur. Wartości pozytywne dodają krzyżyki, wartości negatywne dodają bemole.

```
<time>
  <beats>1</beats>
  <beat-type>4</beat-type>
</time>
```

Powyższy element oznacza metrum. Górny element <*beats*> oznacza liczbę jednostek metrycznych, dolny element <*beat-type*> oznacza podstawową jednostkę metryczną utworu, w tym przykładzie w takcie mamy takty o długości jednej ćwierćnuty.

```
<clef>
  <sign>G</sign>
  <line>2</line>
</clef>
```

<*clef*> definiuje klucz, MusicXML pozwala na użycie kluczy, które już nie są dziś w użyciu. W powyższym kodzie element <*sign*> oznacza typ klucza, a <*line*> linijkę na której się znajduje. G2 to klucz wiolinowy.

```
</attributes>
<note>
```

Koniec definicji atrybutów, czas na zapis nut w takcie.

```
<pitch>
  <step>C</step>
  <octave>4</octave>
</pitch>
```

Element <*pitch*> oznacza wysokość dźwięku nuty. Posiada dwa wymagane elementy <*step*> i <*octave*> oraz jeden opcjonalny <*alter*>. Step oznacza nazwę dźwięku, a octave oktawę. Element alter reprezentuje krzyżyk lub bemol, musi zostać użyty nawet, gdy tonacja została określona w elemencie *fifths*.

```
<duration>1</duration>
```

Nuta użyta w przykładzie to ćwierćnuta, a nasz element <*divisions*> ma wartość 1. Co oznacza, że <*duration*>*1*</*duration*> oznacza, że 1 to długość jednej ćwierćnuty. Gdyby w przykładzie użyto półnuty, element <*duration*> musiałby mieć wartość 2.

```
<type>quarter</type>
```

Element <*type*> jest elementem trochę nadmiernym, ponieważ można wyciągnąć tą informację z elementu *duration* jednak dużo łatwiej pracować jest z obydwoma elementami. Czasem również te dwa elementy nie odpowiadają sobie. Wtedy długość nuty czerpana jest z elementu <*duration*>, a reprezentacja graficzna z elementu <*type*>. Są to jednak bardzo specyficzne przypadki w niewielu utworach.

```
        </note>
      </measure>
    </part>
</score-partwise>
```

Ostatecznie zamykamy wszystkie elementy i nasz utwór składający się z jednej ćwierćnuty jest kompletny.

Jednym z warunków formatu MusicXML jest wymuszona kolejność elementów w strukturze zdefiniowana przez dokument DTD.

3.2 Epoki muzyczne w bazie plików

Początkowo w bazie plików miała znajdować się muzyka z epok od średniowiecza do XX wieku. Jednak ze względu na ograniczony i trudny dostęp do wystarczającej liczby utworów w formacie MusicXML pozostano tylko przy czterech epokach: renesans, barok, klasycyzm oraz romantyzm. Na każdą epokę w bazie przypada około 100 utworów muzycznych różnych artystów z tego każdego okresu. Oto lista artystów z każdej epoki, których utwory były wykorzystane do stworzenia bazy:

- Renesans
  - Josquin des Prez
  - Adrian Willaert

  - Johannes Ockeghem
  - Giovanni Gabrieli
  - Giovanni Pierluigi da Palestrina
- Barok
  - Antonio Vivaldi
  - George Frideric Handel
  - Johann Sebastian Bach
  - Claudio Monteverdi
- Klasycyzm
  - Joseph Haydn
  - Wolfgang Amadeus Mozart
  - Giovanni Battista Pergolesi
  - Nicolà Porpora
  - Luigi Boccherini
- Romantyzm
  - Felix Mendelssohn
  - Ludwig van Beethoven
  - Fryderyk Chopin
  - Franz Liszt
  - Robert Schumann

Rozdział 4

Uczenie maszynowe

4.1 Rodzaje uczenia maszynowego

W tym rozdziale pokrótce przedstawiono trzy rodzaje uczenia maszynowego: uczenie nadzorowane, uczenie nienadzorowane i uczenie przez wzmacnianie.

4.1.1 Uczenie nadzorowane

Główną cechą uczenia nadzorowanego jest fakt, że dane uczące posiadają swoje etykiety klas. Na podstawie cech w danych uczących trenujemy model, który później będzie przewidywał do której klasy należy nieznany obiekt. Jeśli jednak nieznany obiekt będzie należał do nieznanej wcześniej klasy zostanie on niepoprawnie sklasyfikowany.

Ten rodzaj uczenia maszynowego został wybrany do klasyfikacji utworów muzycznych ze względu na posiadaną wcześniej informacje o etykiecie (epoce muzycznej) danych.

4.1.2 Uczenie nienadzorowane

W przeciwieństwie do uczenia nadzorowanego w uczeniu nienadzorowanym nie wiemy do jakiej klasy należą dane uczące. Dzięki takiemu podejściu jesteśmy w stanie poznać strukturę przetwarzanych danych i sklasyfikować je ze względu na podobieństwo.

4.1.3 Uczenie przez wzmacnianie

Uczenie przez wzmacnianie ma na celu stworzenie systemu, który poprawia własną skuteczność na podstawie interakcji ze środowiskiem. System jest nagradzany lub karany w zależności od poprawności wyniku końcowego, w ten sposób dostarczana jest do regulatora informacja zwrotna na podstawie której do modelu wprowadzane są poprawki.

4.2 Algorytmy klasyfikujące

4.2.1 Perceptron

Algorytm klasyfikujący perceptronu działa podobnie do komórki nerwowej, algorytm przemnaża wagi z wartościami wejściowymi i na podstawie wyników określa czy należy zaktualizować wagi czy wynik był poprawny. Wykorzystywana jest do tego funkcja skoku jednostkowego, zwana funkcją skokową Heaviside'a. Algorytm można opisać dwoma krokami:

1. wprowadź wagi o wartościach zerowych lub niewielkich, losowych wartościach,

2. dla każdej próbki uczącej:

   - oblicz wartość wyjściową,
   - zaktualizuj wagi.

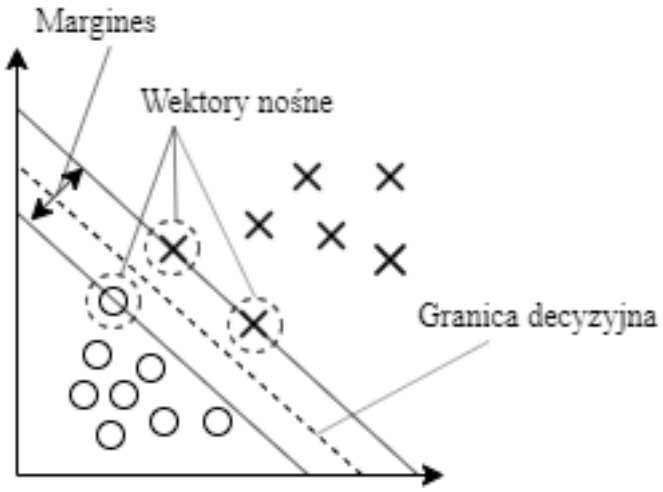
Algorytm kończy swoje działanie, gdy klasy są rozdzielone liniowo. Jeżeli klas nie można rozdzielić liniowo możemy określić maksymalną liczbę epok aktualizacji wag i próg tolerancji nieprawidłowych klasyfikacji.

4.2.2 Regresja logistyczna

Algorytm regresji logistycznej przypomina algorytmy perceptronu jednak tutaj funkcją aktywacji jest funkcja sigmoidalna, której wynik jest interpretowany jako prawdopodobieństwo, że dana próbka przynależy do danej klasy, a ostateczny wynik jest przekształcany za pomocą funkcji skoku jednostkowego.

4.2.3 Maszyny wektorów nośnych

Maszyny wektorów nośnych (ang. support vector machine - SVM) to algorytm, który można uznać za rozwinięcie algorytmu perceptronu. Algorytm perceptronu skupia się na minimalizacji błędów nieprawidłowej klasyfikacji. Z kolei w algorytmie SVM głównym celem jest maksymalizacja marginesu, czyli odległości między najbliższymi próbkami uczącymi (tzw. wektorami nośnymi). Koncepcję tą przybliża rysunek 4.1.



Rysunek 4.1 Margines w algorytmie maszyny wektorów nośnych.

4.3 Przetwarzanie danych

W tym dziale przyjrzymy się sposobom przetwarzania danych wejściowych tak, aby umożliwić skuteczniejsze rozpoznawanie.

4.3.1 Brakujące dane

Niekiedy badane obiekty nie zawierają np. wartości jakiejś cechy, większość algorytmów nie potrafi sobie poradzić w takiej sytuacji. Narzędzia bibliotek pandas oraz scikit-learn pozwalają wykrywać takie sytuacje i podejmować konkretne działania.

- Usuwanie próbek lub cech niezawierających wartości - jednym ze sposobów jest po prostu usunięcie wadliwej próbki lub cechy. Istnieje jednak ryzyko, że usuniemy zbyt wiele próbek lub cech.
- Uzupełnianie brakujących danych - kolejnym rozwiązaniem jest oszacowanie danych na podstawie pozostałych danych. Jedną z najpopularniejszych metod jest imputacja z użyciem średniej. W tej metodzie wyliczana jest średnia na podstawie całej kolumny cech. Zamiast średniej możemy też użyć mediany lub najczęściej występującej cechy.

4.3.2 Cechy kategoryzujące

Często cechy obiektów wejściowych nie są wartościami liczbowymi, w takim przypadku należy rozróżnić dwa rodzaje cech: nominalne i porządkowe. Cechy porządkowe to takie, gdzie możemy ustalić jakąś kolejność, np. rozmiar ubrania XL > L > M > S. Cechy nominalne to na przykład kolor tego ubrania.

W przypadku cech porządkowych wystarczy przypisać wartości liczbowe kolejnym cechom, korzystając z poprzedniego przykładu: XL - 4 L - 3 M - 2 S - 1

Z kolei cechom nominalnych nie można po prostu przypisać wartości liczbowej, jest to często popełniany błąd podczas przetwarzania danych wejściowych. Algorytm będzie zakładał, że kolor np. niebieski jest większy od zielonego. Algorytm może generować poprawne wyniki, chociaż będą one dalekie od optymalnych. Rozwiązaniem jest kodowanie gorącojedynkowe, pod tą nazwą kryje się tworzenie sztucznych cech dla każdej nominalnej wartości. Tabela 4.1 przedstawia kodowanie trzech kolorów dla trzech próbek.

Tabela 4.1 Kodowanie gorącojedynkowe cech nominalnych

indeks	niebieski	czerwony	zielony
0	0	1	0
1	0	0	1
2	1	0	0

Z tabeli 4.1 możemy odczytać, że ubranie o indeksie 0 ma kolor czerwony, indeksie 1 zielony, a indeksie 2 niebieski.

4.3.3 Kodowanie etykiet klas

Zazwyczaj dane wejściowe posiadają etykiety klas zakodowane w postaci nieliczbowej, np. w przypadku tej pracy są to nazwy epok z których pochodzi dany utwór. Dobrym zwyczajem jest przekształcenie nazw etykiet na wartości liczbowe. To jakie wartości przyjmą nie ma znaczenia dla algorytmu, muszą być po prostu unikatowe dla danej etykiety. Zazwyczaj etykiety są numerowane od zera.

4.3.4 Dane uczące i dane testowe

Nasz zbiór danych należy podzielić na dane uczące, czyli te które będą kształtowały model i testowe, czyli przeciw którym będzie model sprawdzany. Rezygnujemy wtedy z części informacji, więc nie możemy umieszczać zbyt wielu próbek w zbiorze testowym. Nie ma dokładnej proporcji podziału danych i zależy ona od wielkości zbioru. Najczęściej są to stosunki 60:40, 70:30, 80:20 lub w przypadku wużych zbiorów są spotykane wartości takie jak 90:10 lub 99:1.

4.3.5 Skalowanie cech

Skalowanie cech polega na sprowadzeniu wszystkich cech do jednej skali co przyspiesza działanie algorytmów uczenia maszynowego. Najpopularniejsze dwie metody skalowania cech to normalizacja i standaryzacja. Aby przeprowadzić normalizację wystarczy

przeskalować każdą kolumnę cech wobec wartości krańcowych ze wzoru:

$$x_{norm}^{(i)} = \frac{x^{(i)} - x_{min}}{x_{max} - x_{min}} \tag{4.1}$$

Standaryzację kolumny można przeprowadzić wykorzystując wzór:

$$x_{std}^{(1)} = \frac{x^{(i)} - \mu_x}{\sigma_x} \tag{4.2}$$

4.3.6 Dobór cech

Częstym problemem jest tak zwane nadmierne dopasowanie modelu, dzieje się tak, gdy model znacznie lepiej klasyfikuje dane ze zbioru uczącego niż dane rzeczywiste. Taki model cechuje duża wariancja. Przyczynami takich wyników jest zbyt duża złożoność modelu, a problem ten można rozwiązać poprzez: większą ilość danych uczących, karę za złożoność, prostszy model lub zmniejszenie wymiarowości danych.

Rozdział 5

Implementacja aplikacji

5.1 Środowisko

Aplikacja została napisana przy użycia języka programowania Python w wersji 3.6 oraz zewnętrznych bibliotek do tego języka. Poza standardowymi bibliotekami języka Python użyto:

- *NumPy* - biblioteka umożliwiająca szybkie operacje na macierzach wielowymiarowych,
- *Pandas* - dostarcza zoptymalizowane i łatwe w obsłudze struktury danych, które są kompatybilne z narzędziami biblioteki *NumPy*,
- *scikit-learn* - biblioteka umożliwiająca klasyfikację obiektów na podstawie uczenia maszynowego, jest kompatybilna z narzędziami dostarczanymi przez *NumPy* oraz *Pandas*,
- *music21* - zestaw narzędzi do analizy różnych formatów muzycznych, między innymi MIDI oraz MusicXML, pozwala również na pisanie własnych utworów w języku Python.

5.2 Biblioteka music21

Dzięki prostemu interfejsowi biblioteki *music21* w bardzo łatwy sposób można odczytywać informacje z plików MusicXML.

```
import music21

stream = music21.converter.parse("C:\\sciezka\\do\\pliku.mxl")
```

Powyższy kod tworzy obiekt typu *Stream* zawierający wszystkie informacje przechowywane w pliku MusicXML, a także udostępnia metody pozwalające na działanie na tym obiekcie.

5.2.1 Parametryzacja plików

Następujące parametry zostały pozyskane z plików muzycznych i były używane do stworzenia danych wejściowych dla algorytmów uczenia maszynowego:

- średnia wysokość dźwięku,
- średnia wysokość dźwięku ważona czasem trwania dźwięku,
- odchylenie standardowe wysokości dźwięku,
- średnia długość dźwięku,
- odchylenie standardowe długości dźwięku,
- znormalizowana liczba nut ze względu na wysokość dźwięku,
- znormalizowana liczba nut ze względu na czas trwania dźwięku.

5.2.2 Implementacja parametryzacji plików

Została napisana klasa *StreamAnalyzer*, która przyjmuje dwa argumenty podczas tworzenia obiektu - *stream* oraz *era*. Argument *stream* to obiekt typu *music21.Stream*, czyli Python'owa reprezentacja pliku muzycznego, a *era* oznacza epokę z której ten plik pochodzi (argument wymagany do tworzenia bazy danych). Klasa *StreamAnalyzer* zawiera pola, z wyżej wymienionymi parametrami, które są pozyskiwane bezpośrednio z obiektu *Stream* przy pomocy metody klasy *StreamAnalyzer* - *extract_parameters()*, która znajduje się poniżej. Lista zawierająca wszystkie nuty w utworze tworzona jest przez prywatną metodę klasy *StreamAnalyzer* *_flatten_stream()*, która łączy wszystkie części utworu w jeden
i zbiera wszystkie nuty w jedną listę, która dalej jest przetwarzana.

```
def extract_parameters(self) -> None:
    """
    Method that calculates parameters from list of notes
    """
    pitch_frequencies = [note.pitch.frequency
                         for note in self.notes_in_stream]
    weighted_pitch_frequencies = \
```

```
        [note.pitch.frequency * note.duration.quarterLength
        for note in self.notes_in_stream]
    notes_durations = \
      [note.duration.quarterLength
        if not isinstance(
        note.duration.quarterLength, fractions.Fraction)
        else note.duration.quarterLength.numerator / \
            note.duration.quarterLength.denominator
        for note in self.notes_in_stream]
    self.avg_pitch_freq = np.average(pitch_frequencies)
    self.weighted_avg_pitch_freq = \
        np.average(weighted_pitch_frequencies)
    self.pitch_std = np.std(pitch_frequencies)
    self.avg_note_duration = np.average(notes_durations)
    self.note_duration_std = np.std(notes_durations)
    self.notes_by_pitch = Counter(pitch_frequencies)
    self.notes_by_duration = Counter(notes_durations)
    self.name = self.stream.filePath.stem
```

Metoda nie przyjmuje żadnych argumentów. Na początku tej metody tworzone są listy wysokości dźwięków w utworze, ważonych wysokości dźwięków w utworze, gdzie wagą jest długość dźwięku oraz lista długości dźwięków w utworze. W kolejnym kroku obliczane i przypisywane do pól obiektu są parametry. Obliczenia wykonywane są przy pomocy biblioteki *NumPy* oraz obiektu *Counter* należącego do biblioteki standardowej *collections*. Obiekt *Counter* pozwala w prosty sposób policzyć liczbę występowania danej wartości w liście. Na koniec do obiektu przypisywana jest też nazwa pliku.

W ten sposób tworzone są obiekty zawierające już tylko przydatne informacje.

5.3 Tworzenie DataFrame oraz bazy CSV

Kolejnym etapem jest przetworzenie listy obiektów typu *StreamAnalyzer* zawierających informacje o utworach muzycznych w jedną pełną bazę danych. Służy do tego klasa znajdująca się w pliku *data_parser.py* o tej samej nazwie *DataParser*. Metody klasy *DataParser* pozwalają na stworzenie jednego dużego słownika zawierającego informacje o utworach, przekształcenie go tak, aby miał tylko twa wymiary, ponieważ parametry "Znormalizowana liczba nut ze względu na wysokość dźwięku" oraz "Znormalizowana liczba nut ze względu na czas trwania dźwięku" wciąż są przechowywane w formie słowników. Przekształcenie pozwala na stworzenie nowych cech na podstawie każdego klucza

z owych słowników - zamiast dwóch parametrów tworzy się ich wiele w zależności od tego ile rodzajów nut było w omawianych dwóch parametrach. Robi to metoda klasy *DataParser* o nazwie *flatten_ dictionaries()*, która jednocześnie normalizuje liczbę nut względem wszystkich nut w utworze, tak żeby długość utworu nie miała większego wpływu na wyniki.

Metoda *flatten_ dictionaries()* zwraca słownik, z którego tworzony jest obiekt typu *DataFrame* biblioteki *pandas* za pomocą metody *create_ data_ frame()*. Obiekt *DataFrame* może być zapisany (i równie prosto odczytany) bezpośrednio do pliku CSV (ang. *comma seperated values*, czyli formatu pozwalającego do przechowywania uporządkowanych danych w plikach tekstowych. Tak przygotowane dane można przekazać do algorytmów uczenia maszynowego.

5.4 Wybieranie parametrów

Jako, że parametrów dla każdego utworu na tym etapie jest bardzo dużo - około 110, należy zatem wybrać te najbardziej znaczące. Z pomocą przychodzi biblioteka *scikit-learn* posiadająca klasę *RandomForestClassifier* w module *ensemble*. Klasa ta dzięki metodzie *fit()* pozwala na estymowanie ważności wag zbioru cech dzięki temu zwiększając celność modelu i kontrolę nadmiernego dopasowania. Na rysunku 5.1 przedstawiono fragment listingu konsoli przedstawiającego 25 pierwszych cech i ich wagi estymowane przez klasę *RandomForestClassifier*.

We wszystkich powtórzeniach na pierwszym miejscu zawsze znajdywały się cechy:

- 2.0 - ilość nut o długości dwóch pełnych nut,
- pitch_std - odchylenie standardowe wysokości dźwięku,
- avg_note_duration - średnia długość dźwięku,
- 4.0 - ilość nut o długości czterech pełnych nut.

Na podstawie tych cech ręcznie została przetworzona baza danych i stworzony nowy plik CSV posiadający tylko te cechy. Baza ta została później użyta do wytrenowania ostatecznego modelu.

```
    Cecha                          Waga
 1) 2.0                            0.048387
 2) pitch_std                      0.039110
 3) avg_note_duration              0.037821
 4) 4.0                            0.031865
 5) 783.990871963499               0.028927
 6) note_duration_std              0.026779
 7) 1.5                            0.026598
 8) 0.5                            0.025983
 9) avg_pitch                      0.025233
10) 0.25                           0.024024
11) weighted_avg_pitch_freq        0.023470
12) 698.456462866008               0.019106
13) 0.75                           0.018011
14) 277.182630976872               0.016926
15) 138.59131548843592             0.016334
16) 311.1269837220808              0.015667
17) 523.2511306011974              0.015426
18) 123.4708253140309              0.014925
19) 164.81377845643485             0.014915
20) 739.988845423269               0.014700
21) 261.6255653005985              0.014564
22) 293.66476791740746             0.014136
23) 659.2551138257401              0.014047
24) 207.65234878997245             0.013893
25) 587.3295358348153              0.013691
```

Rysunek 5.1 Fragment konsoli pokazujący znaczenie wagi cech.

5.5 Trenowanie modelu

Posiadając przygotowane dane w prosty sposób można było przygotować model. Biblioteka *scikit-learn* w prosty sposób pozwala podzielić zestaw danych na testowy i treningowy przy pomocy funkcji *train_test_split()*.

```
params_train, params_test, labels_train, labels_test = \
    train_test_split(params, labels, test_size=test_size,
                     random_state=random_state)
```

Argument test_size odpowiada za stosunek podziału, domyślnie 60:40, gdzie 60% próbek to próbki testowe. Stosunek ten można zmieniać podczas trenowania modelu używając odpowiedniego argumentu wiersza poleceń, o którym będzie w dalszej części pracy. W kolejnym kroku należy przeprowadzić standaryzację danych testowych i uczących zgodnie ze wzorem 4.2 za co odpowiada klasa *StandardScaler* biblioteki *scikit-learn*.

```
sc = StandardScaler()
sc.fit(params)
params_train_std = sc.transform(params_train)
params_test_std = sc.transform(params_test)
```

Zmienne *params_train_std* oraz *params_test_std* są posiadają ustandaryzowane wartości wykorzystane do uczenia modelu.

```
svc = SVC(kernel=kernel, random_state=random_state, gamma=gamma,
          C=C)
svc.fit(X=params_train_std, y=labels_train)

predictions = svc.predict(params_test_std)
```

Powyższy listing przedstawia inicjalizację obiektu SVC biblioteki *scikit-learn* używającą algorytmu maszyny wektorów nośnych. Funkcja *fit()* jako argumenty przyjmuje ustandaryzowane wartości stworzone w poprzednim kroku oraz odpowiadające im etykiety danych oraz trenuje model. Zmienna *predictions* to lista przewidzianych przez model etykiet na podstawie danych *params_test_std* użytych jako argument w metodzie *predict*. Dzięki porównaniu jej ze zmienną *labels_test* możemy sprawdzić skuteczność modelu dla danych testowych. Na koniec drukujemy do konsoli wyniki trenowania, ile było próbek testowych, ile próbek sklasyfikowano niepoprawnie oraz jakie są dokładności modelu dla danych testowych i uczących.

5.6 Interfejs wiersza poleceń

Aplikacja opiera się o interfejs wiersza poleceń (ang. *command line interface*) i udostępnia następujące komendy:

- –csv-path [PATH] - jedyny wymagany argument, wskazuje ścieżkę do pliku CSV zawierającego przekształconą bazę danych,
- –mxml-files [PATH] - ścieżka do folderu z plikami MusicXML, tworzy nową bazę danych z plików MusicXML w miejscu wskazanym przez argument –csv-path,
- –train - nie przyjmuje żadnych argumentów, jeśli jest podany na nowo tworzony jest model,
- –test-size [NUMBER] - ułamek grupy testowej z bazy danych, domyślnie 0.4,
- –kernel [STRING] - wybór jądra modelu, domyślnie "rbf",

- –gamma [NUMBER] - obszar graniczny strefy Gauss'a, tym większa gamma tym mniej sztywne są granice decyzyjne modelu, domyślnie 0.1,
- -C [NUMBER] - odwrotność parametru regularyzacji, zapobiega zbyt dużej wartości wag cech, domyślnie 1.0,
- –random-state [STRING or INTEGER] - generator liczb pseudolosowych, jeśli podana jest liczba stała, użyta jest jako ziarno generatora liczb losowych, domyślnie użyty jest generator biblioteki *NumPy*,
- –save-model [PATH] - ścieżka do zapisu modelu, model powinien być zapisany z rozszerzeniem *.joblib*,
- –classify [PATH] - ścieżka do zapisanego modelu, jeśli został podany argument –train użyty zostanie aktualny model z pamięci,
- -h, –help - pomoc, wyświetla informacje o dostępnych argumentach.

5.6.1 Przykłady użycia

```
py mxml_recognizer.py \
        --csv-path .\bazy_danych\nowa_baza.csv \
        --mxml-files .\pliki_musicxml
```

Powyższa komenda tworzy nową bazę danych w pliku *nowa_baza.csv* na podstawie plików znajdujących się w folderze *pliki_musicxml*.

```
py mxml_recognizer.py \
        --csv-path .\bazy_danych\nowa_baza.csv \
        --train \
        --save-model .\modele\nowy_model.joblib
```

Trenuje model na podstawie istniejącej bazy danych *nowa_baza.csv* oraz zapisuje go do pliku *nowy_model.joblib*.

```
py mxml_recognizer.py \
        --csv-path .\bazy_danych\testowa_baza.csv \
        --classify .\modele\nowy_model.joblib
```

Klasyfikuje przetworzone na format CSV pliki z pliku *testowa_baza.csv*.

Rozdział 6

Podsumowanie

Celem pracy inżynierskiej było wykonanie aplikacji do automatycznej klasyfikacji utworów muzycznych ze względu na przynależność do danej epoki, cel został wykonany jednak wyniki działania aplikacji nie są idealne i jest na pewno sporo miejsca na poprawę.

6.1 Wyniki

6.2 Wnioski

WIP

6.3 Możliwości rozwoju

BRAK