­­МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра вычислительных технологий**

**КУРСОВАЯ РАБОТА**

**КЛАССИФИКАЦИЯ МУЗЫКАЛЬНЫХ КОМПОЗИЦИЙ ПО ИСПОЛНИТЕЛЯМ НА ОСНОВЕ МАШИННОГО ОБУЧЕНИЯ**

Работу выполнил Н.С. Лысенко

(подпись, дата) (инициалы, фамилия)

Факультет компьютерных технологий и прикладной математики 3 курс

Направление 02.03.03 – «Математическое обеспечение и администрирование информационных систем»

Научный руководитель

доц., канд.т.н. \_\_\_\_\_\_\_Т.А. Приходько

(подпись, дата) (инициалы, фамилия)

Нормоконтролер

ассистент \_\_\_\_\_\_\_\_\_\_\_\_\_\_ А.А. Климец

(подпись, дата) (инициалы, фамилия)

Краснодар 2019

**РЕФЕРАТ**

Курсовая работа содержит 46 страниц, 16 рисунков, 10 источников, 7 формул.

КЛАССИФИКАЦИЯ, СКРЫТЫЕ МАРКОВСКИЕ МОДЕЛИ, ЗВУКОВЫЕ СИГНАЛЫ, МЕЛ-КЕПСТРАЛЬНЫЕ КОЭФФИЦИЕНТЫ, ОБУЧЕНИЕ КЛАССИФИКАТОРА

Объектом исследования является классификатор музыкальных композиций по исполнителям, его обучение и принцип работы.

Цель курсовой работы – изучение методов машинного обучения и средств языка Python для классификации музыкальных композиций по исполнителям.

Результатом проделанной работы является знакомство с реализацией алгоритмов машинного обучения, методами извлечения информации из звуковых сигналов, понимание работы классификатора песен по исполнителям и выявлением особенностей его обучения.

Реализация классификатора музыкальных произведений по исполнителям будет осуществляться на языке программирования Python – высокоуровневом языке программирования с множеством различных библиотек, предназначенных для реализации алгоритмов машинного обучения, с помощью среды разработки PyCharm и Google Colaboratory.

**СОДЕРЖАНИЕ**

[Введение 4](#_Toc27128977)

[1 Обзор методологий, применяемых для решения задач классификации 6](#_Toc27128978)

[1.1 Формальная постановка задачи классификации 6](#_Toc27128979)

[1.2 Использование Преобразования Фурье в задачах классификации 6](#_Toc27128980)

[1.3 Дискретное преобразование Фурье 7](#_Toc27128981)

[1.4 Мел-частотные кепстральные коэффициенты 8](#_Toc27128982)

[1.5 Скрытые марковские модели 13](#_Toc27128983)

[2 Реализация алгоритма классификации. Описание используемых методов, функций и библиотек 16](#_Toc27128984)

[2.1 Возможности Python по анализу данных 16](#_Toc27128985)

[2.2 Извлечение информации из музыки 18](#_Toc27128986)

[3 Экспериментальное исследование обучающих данных 23](#_Toc27128987)

[3.1 Описание исходных данных 23](#_Toc27128988)

[3.2 Алгоритм решения задачи 23](#_Toc27128989)

[3.3 Первичный визуальный анализ 24](#_Toc27128990)

[3.4 Обучение классификатора 28](#_Toc27128991)

[4 Анализ полученных результатов 32](#_Toc27128992)

[Заключение 34](#_Toc27128993)

[Список использованных источников 35](#_Toc27128994)

[Приложение 36](#_Toc27128995)

# **ВВЕДЕНИЕ**

Классификация – это один из разделов машинного обучения, задача которого заключается в присвоении меток объектам. Например, если объектами являются фотографии, то, как правило, метками может быть содержимое фотографий, то есть, содержит ли изображение автомобили или нет, изображен ли взрослый человек или ребёнок, какой породы собака изображена на фотографии. В основном, существуют уже заранее известные для определенного набора объектов метки, а также взаимоисключающие метки. Имея подобный набор данных, требуется построить алгоритм, который позволит автоматически расставлять метки на произвольных объектах того же типа, что были в начальном множестве объектов, тем самым классифицируя произвольный объект из исходного множества. Данные об объектах классификации могут представляться в разных формах. Наиболее частым случаем является представление объекта через его признаковое описание, однако существуют и другие виды представления данных.

В настоящее время задача распознавания и поиска музыкальных композиций становится весьма актуальной в связи с увеличением числа интернет сервисов с музыкальной тематикой и ускоренным ростом их популярности. Нейросетевые классификаторы уже являются широко используемыми, потому что они обладают способностью обучаться на ограниченном множестве входных данных и имеют эффективные алгоритмы обучения.

Задача точной идентификации жанра или исполнителя музыкальной композиции является крайне сложной не только для компьютера, но и для человека, а он является самым совершенным инструментом идентификации на данный момент.

Цель работы заключается в изучении алгоритмов машинного обучения для классификации музыкальных композиций по исполнителям средствами языка программирования Python.

Курсовая работа состоит из четырёх глав, введения, заключения, списка использованных источников и приложения.

Первая глава работы содержит общие теоретические сведения о рассматриваемой задаче классификации и алгоритмах её решения.

Вторая глава данной работы посвящена описанию реализации алгоритмов классификации.

В третьей главе приводятся требования к исходным данным задачи и представлению результатов решения.

Четвертая глава посвящена анализу полученных результатов.

Заключение содержит основные выводы.

В Приложении приведен код классификатора.

# Обзор методологий, применяемых для решения задач классификации

# Формальная постановка задачи классификации

Пусть имеется множество объектов *X*. Объектами могут быть или точки на плоскости, или фотографии, или музыкальные композиции. Пусть также существует конечное множество пронумерованных меток *Y*. Мы будем отождествлять метки с их номерами. Тогда, например, исходя из наших определений, *Y={red,green,blue}* будет обозначаться как *Y={1,2,3}*. Дополнительно, у нас есть входная выборка:

*D={(xi , yi), xi X, yi Y, i=1..N}* (1)

Это те самые размеченные примеры, на которых мы и будет осуществляться обучение классификатора проставлять метки автоматически. Классы всех объектов точно неизвестны, поэтому можно считать, что класс объекта − это случайная величина, которую для простоты также будем обозначать *y*. Например, ноктюрн Шопена может классифицироваться как композиция в стиле классика с вероятностью 0.99 и как композиция в стиле рок с вероятностью 0.01. Таким образом, чтобы определить принадлежность объекта определённому классу, нужно знать условное распределение случайной величины *y* на данном объекте *x* − *p(y|x).* [8]

Задача нахождения *p(y|x)* при данном множестве меток *Y* и данном наборе размеченных примеров (1)называется задачей классификации.

# Использование Преобразования Фурье в задачах классификации

Когда речь идёт о классификации звуковых сигналов, во многих случаях удобно представлять сигнал в виде спектра, или нарисовать спектрограмму сигнала. Спектрограмма − это наглядное представление встречающихся частот. Пусть есть некоторый сигнал длительностью T секунд. Математически он является некоторой непрерывной непериодической функцией *f(x)*, заданной на отрезке *{0, T}* (*X* в данном случае – время). Такую функцию всегда можно представить в виде суммы гармонических функций (синусоид или косинусоид) вида:

(2)

где *k* – номер тригонометрической функции (номер гармонической составляющей), *τ* – отрезок, где функция определена (длительность сигнала), *Ak* – амплитуда *k*-ой гармонической составляющей, *θk* – начальная фаза *k*-ой гармонической составляющей.

Данное представление функции называется рядом Фурье и является математической основой спектрального анализа сигналов. [7]

# Дискретное преобразование Фурье

В настоящее время сигналы для хранения оцифровываются и содержатся в памяти в виде набора отсчётов. Одним из основных параметров АЦП является максимальная частота дискретизации — частота взятия отсчетов непрерывного во времени сигнала при его дискретизации (измеряется в Гц). Для того, чтобы вычислить спектр сигнала по его дискретным отсчетам используется дискретное преобразование Фурье. Непрерывный сигнал имеет спектр, «по определению» ограниченный максимальной частотой, которая обычно обозначается как и равна меньшей половине частоты дискретизации (Fd). Следовательно, спектр дискретного сигнала может быть представлен как сумма конечного числа гармоник. Согласно теореме Котельникова, максимальная частота гармоники должна быть такой, чтобы на неё приходилось как минимум два отсчета, поэтому число гармоник равно половине числа отсчетов дискретного сигнала. То есть если в выборке имеется *N* отсчетов, то число гармоник в спектре будет равно *N/2*. Дискретное преобразование Фурье (ДПФ) будет выглядеть следующим образом:

(3)

где *k* – номера отсчетов сигнала, *s* – номера спектральных составляющих; , *k=0,1,2,…,N-1* и .

Величина *s* показывает количество полных колебаний гармоники на периоде *Т* (длительности сигнала). Время в ДПФ имеет дискретный характер, и число гармоник ограничено величиной *N/2*. Дискретное преобразование Фурье используется для нахождения амплитуд и фаз гармоник численным методом. Таким образом, БПФ (Быстрое Преобразование Фурье) и ДПФ используются, чтобы представить сигнал в виде ряда значений периодической функции, а чтобы получить «правильный» спектр сигнала, являющегося суммой нескольких синусоид с разными периодами, необходимо, чтобы на периоде измерения сигнала укладывалось целое число периодов каждой синусоиды. [1]

# Мел-частотные кепстральные коэффициенты

Для задач распознавания речи и звуковых сигналов используется метод, основанный на измерении скорости изменения спектральных характеристик сигнала, которые также называются мел-частотными кепстральными коэффициентами (MFCC). Эти коэффициенты – представление энергии сигнала в виде особого спектра, из которого путём различных фильтраций и преобразований удалены незначительные для человеческого уха компоненты и шумы. Спектр проецируется на так называемую мел-шкалу. Изначально сигнал делится на пересекающиеся отрезки по 20-40 мс (предполагается, что на этих отрезках частоты сигнала не меняются слишком сильно) на которых рассчитываются MFCC. Подобное представление сохраняет волновую «природу» сигнала и позволяет выделить наиболее значимые для человеческого уха частоты. К тому же, для каждого сигнала расположение данных коэффициентов на мел-шкале уникально, что позволяет использовать этот мел-частотный спектр в задачах распознавания речи и классификации музыкальных композиций в качестве важной характеристики каждого рассматриваемого сигнала в процессе обучения классификатора.

Вычисление мел-частотных кепстральных коэффициентов включает в себя следующие шаги [5]:

* 1. Необходимо разделить исходный сигнал на кадры. Их размер обычно выбирается в диапазоне от 10 до 40 мс. Кадры накладываются друг на друга. В данном случае выберем длину кадра, например, 10мс, а наложение – 5мс. Так как в основном мел-кепстральные коэффициенты используются в выделении фонем в задачах автоматического распознавания речи, в качестве примера возьмём график временного представления слова «один», изображённый на рисунке 1.

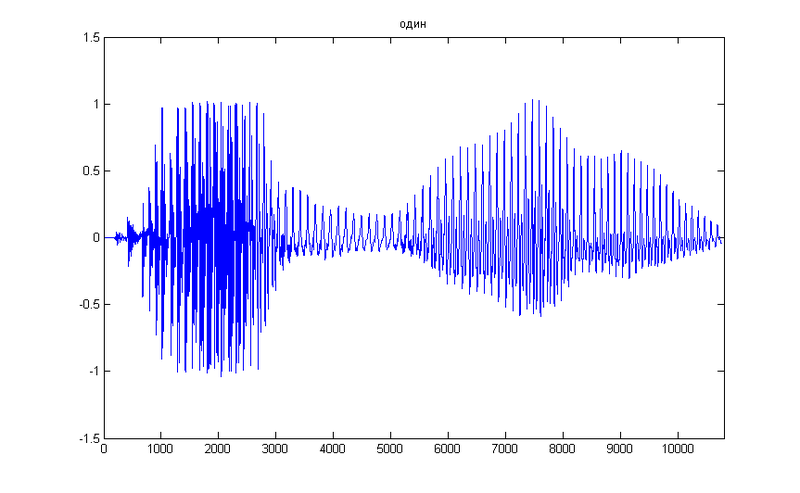


Рисунок 1 – График зависимости амплитуды сигнала от времени для слова «один» (вертикальная ось – частота, горизонтальная – время в секундах).

* 1. Полагаем, что звуковой сигнал конечен и не является периодическим, поэтому из-за разрывов на его концах, когда применяется преобразование Фурье, может произойти так называемый эффект утечки. В дискретной области так называют эффекты, связанные с искажениями в Фурье анализе, вызванные конечностью выборки. В случае периодического дискретного процесса этих эффектов удается избежать, если специальным образом выбирать шаг дискретизации и длину временного ряда. То есть, можно контролировать «ширину» этой утечки или, концентрируя её вокруг главной частоты (сильно “размывается” спектр, зато не мешает соседним частотам), или, растягивая её на бесконечности (но в этом случае размытие пиков уменьшается, а «шум» сильно растёт и вследствие этого растёт погрешность измерения амплитуды отдельных частот). Как правило, чтобы снизить влияние эффекта утечки на результат, к каждому кадру применяется оконная функция (весовая функция). Часто «конечный» сигнал, используемый для преобразования Фурье, в реальности является частью возможно бесконечного сигнала, такого, как синусоида, например. В этой ситуации применяют дополнение конечного отрезка нулями: считают, что исходный сигнал имеет бесконечно большую длину, но затем умножается на некоторую взвешивающую функцию — “окно”, обращающуюся в ноль вне доступного для измерения отрезка. В простейших случаях роль «окна» играет прямоугольная функция, с помощью которой мы, по сути, дополняем отрезок слева и справа бесконечным количеством нулей. В данном случае используем оконную функцию Хемминга:

*w(n) = 0.54 – 0.46cos(2)* (4)

* 1. К каждому кадру применяется преобразование Фурье, получая тем самым спектр сигнала. Далее вычисляется периодограмма – оценка спектральной плотности мощности сигнала:

*Pi (k) = |Yi(k)|2/N* (5)

* 1. Полученное представление сигнала в частотной области разбивают на диапазоны с помощью блока треугольных пересекающихся фильтров, расположенных наиболее плотно в области низких частот. Границы фильтров рассчитывают в мел-шкале. Данная шкала является результатом исследований по способности человеческого уха к восприятию звуков на различных частотах. Количество таких фильтров - 26. Для расчёта фильтров обычно выбираются верхняя и нижняя частоты, а затем осуществляется переход от частотной шкалы к мел-шкале по формуле (6):

*M(f) = 1127\*ln(1+f/700)* (6)

Между полученными значениями на мел-шкале выбираются точки, расположенные линейно, как правило, для 26 фильтров — 28 точек. После этого переход обратно в частотную шкалу осуществляется по обратной формуле 7:

*F(m) = 700(exp(m/1125) - 1)*  (7)

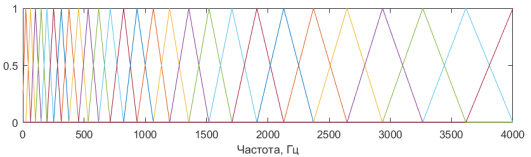
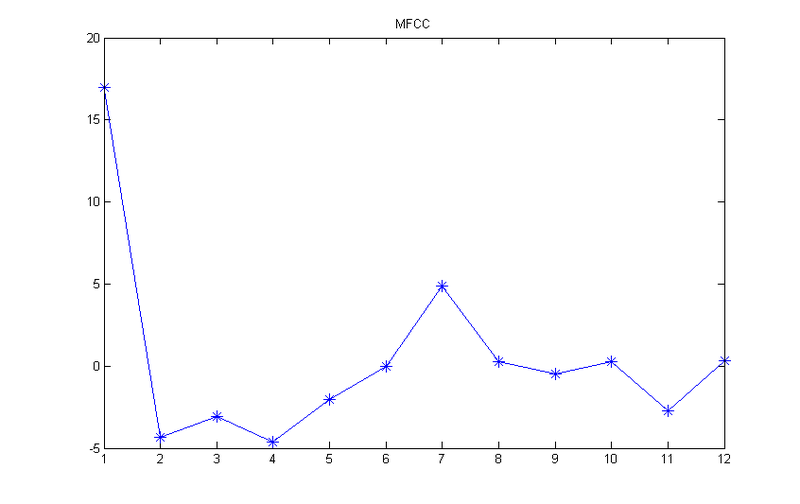


Рисунок 2 – График частотной шкалы

Из графика видно, что окна сжимаются в большей степени в области низких частот (рисунок 2). Далее простым перемножением векторов спектра сигнала и оконной функции находится энергия сигнала, которая попадает в каждое из анализируемых окон. Получаем некоторый набор коэффициентов. Но это ещё не мел-кепстральные коэффициенты, которые мы хотим найти. Полученные значения энергии сигнала возводятся в квадрат и логарифмируются, после чего применяется дискретное косинусное преобразование. В итоге, имеем 26 коэффициентов для каждого кадра (окна) — это и есть мел-частотные кепстральные коэффициенты. В задачах распознавания речи обычно используются первые 13-16 из них, как наиболее информативные для речевого сигнала.

В результате последовательность коэффициентов выглядит так, как показано на рисунке 3.

Рисунок 3 – График расположения спектральных частотных коэффициентов на мел-шкале

Таким образом, мы получили небольшой набор значений, преимущество которого при распознавании звукового и речевого сигнала в том, что достаточно брать около 16 коэффициентов на каждый кадр вместо сотен или тысяч, как было бы в случае обычного преобразования Фурье. Экспериментальным путем выяснено, что для выделения этих коэффициентов в песнях лучше брать по 30-40 компонент.

# Скрытые марковские модели

Чтобы разобраться в том, что из себя представляют скрытые марковские модели, для начала ознакомимся с понятием Марковского процесса. Марковскийпроцесс — случайный процесс, эволюция которого после любого заданного значения временного параметра t не зависит от эволюции, предшествовавшей t, при условии, что значение процесса в этот момент фиксировано («будущее» процесса зависит от «прошлого» лишь через «настоящее»). Например, по оси абсцисс случайным образом перемещается точка. Причём её перемещение происходит так: в момент времени *t=0* она находится в начале координат и остается там в течение одной секунды. Через секунду бросается монета — если выпал герб, то точка перемещается на одну единицу длины вправо, если решка — влево. Через секунду снова бросается монета и производится такое же случайное перемещение, и так далее. Процесс изменения положения точки является случайным процессом с дискретным временем *t=0, 1, 2,* … и счётным множеством состояний. Такое перемещение точки и представляет собой марковский процесс, потому что следующее состояние точки зависит только от текущего состояния и не зависит от прошлых состояний. [6]

Теперь перейдём непосредственно к определению скрытых марковских моделей. Скрытая марковская модель (СММ) — статистическая модель, имитирующая работу процесса, похожего на марковский процесс с неизвестными параметрами, и задачей является разгадывание неизвестных параметров на основе наблюдаемых. Полученные параметры могут быть использованы в дальнейшем анализе, например, для распознавания образов.

В обычном марковском процессе состояние видимо наблюдателю, поэтому имеется единственный параметр – вероятности переходов. В скрытой же Марковской модели мы можем следить лишь за переменными, на которые оказывает влияние данное состояние. Каждое состояние имеет вероятностное распределение среди всех возможных выходных значений. Поэтому последовательность символов, сгенерированная СММ, даёт информацию о последовательности состояний.

Структура скрытых марковских моделей в общем случае может быть представлена как конечный автомат с перепадами между любыми парами состояний, как показано на рисунке 4.

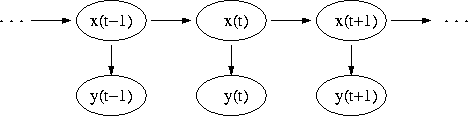


Рисунок 4 – Конечный автомат, реализующий структуру скрытых марковских моделей

Здесь случайная переменная *x ( t ) {\displaystyle x(t)} x(t)* представляет собой значение скрытой переменной в момент времени *t {\displaystyle t} t*. Случайная переменная *y ( t ) {\displaystyle y(t)} y(t)* — это значение наблюдаемой переменной в момент времени *t {\displaystyle t} t*. Стрелки на диаграмме символизируют некоторые условные зависимости. Из диаграммы становится ясно, что значение скрытой переменной x ( t ) x(t) зависит только от значения скрытой переменной *x ( t − 1 ) {\displaystyle x(t-1)} x(t-1)* в момент t − 1 времени *t-1*. Это называется свойством Маркова. Хотя в то же время значение наблюдаемой переменной *y ( t ) {\displaystyle y(t)} y(t)* зависит только от значения скрытой переменной *x ( t ) {\displaystyle x(t)} x(t)*.

Основное применение скрытые модели Маркова получили в области распознавания рукописного текста, речи, движений, сигналов.

В качестве наблюдаемых результатов в нашей задаче классификации выступают мел-кепстральные коэффициенты (MFCC) для каждого фрагмента сигнала. Для нахождения неизвестных параметров СММ используется Алгоритм Баума-Велша, именно он и занимается обучением модели.

# Реализация алгоритма классификации. Описание используемых методов, функций и библиотек

# Возможности Python по анализу данных

Для анализа данных поставленной задачи и реализация алгоритма классификации используется язык программирования Python, для которого имеется большое количество библиотек и фреймворков для реализации машинного обучения, анализа данных и их визуализации. В работе используются облачный сервис Google Colaboratory и интерпретатор PyCharm. Google Colaboratory — это недавно появившийся облачный сервис, направленный на упрощение исследований в области машинного и глубокого обучения. Используя Colaboratory, можно получить удаленный доступ к машине с подключенной видеокартой, что приносит большую пользу, когда приходится обучать глубокие нейронные сети. Можно утверждать, что она является некоторым аналогом google-документов для Jupyter Notebook. В Google Colaboratory содержатся практически все библиотеки Python, необходимые для написания программ визуализации данных, для сложных математических расчётов, при создании различных обучаемых моделей, глубоких нейронных сетей и многого другого. Также в ней присутствует функция разбиения написанного кода на независимые блоки, вследствие чего появляется возможность запуска отдельных частей программы, что очень удобно для выявления ошибок в коде и при отладке программы. Этот функционал получилось реализовать, потому что Python является интерпретируемым языком (то есть команды исполняются построчно).

Для визуализации данных в работе используется библиотека matplotlib.

В качестве библиотеки для работы со звуком было решено выбрать библиотеку Librosa. Она позволяет создать полноценную систему извлечения музыкальной информации. Также её преимущества в том, что Librosa предназначена именно для обработки музыки и хорошо документирована.

Машинное обучение осуществляется с помощью библиотеки Scikit-Learn, в которой реализовано большое количество алгоритмов машинного обучения, например:

* загрузка и нормализация данных;
* отбор признаков;
* реализация алгоритма логистической регрессии (объект LogisticRegression), которая чаще всего используется для решения задач классификации (и бинарной, и многоклассовой). Достоинство данного алгоритма в том, что на выходе для каждого объекта мы получаем вероятность принадлежности его определённому классу;
* реализация алгоритма Наивной Байесовской классификации, основной задачей которого является восстановление плотностей распределения данных обучающей выборки. Зачастую этот метод дает хорошее качество в задачах именно многоклассовой классификации;
* реализация метода k-ближайших соседей (kNN), который часто используется как составная часть более сложного алгоритма классификации. Например, его оценку можно использовать как признак для рассматриваемого объекта;
* оптимизация параметров алгоритма и др. [2]

В ходе изучения возможностей библиотеки выяснилось, что, начиная с версии 0.17, которая является уже устаревшей, Scikit-Learn не содержит необходимого модуля реализации метода скрытых марковских моделей и их обучения, который понадобится для решения поставленной задачи классификации. Теперь он существует как полноценная библиотека – HMMlearn (Hidden Markov Models), поэтому его нужно устанавливать отдельно.

Также в работе используется библиотека NumPy — библиотека с открытым исходным кодом для языка программирования Python. Основной возможностью этой библиотеки является поддержка многомерных массивов (включая матрицы) и поддержка высокоуровневых математических функций, предназначенных для работы с многомерными массивами. Эта библиотека использует параллельные вычисления, и содержит алгоритмы для решения задач линейной алгебры. Numpy входит в библиотеку для научных вычислений SciPy.

Возможности библиотеки SciPy достаточно обширны. С помощью этого пакета можно:

* искать минимумы и максимумы функций;
* вычислять интегралы функций;
* обрабатывать сигналы и изображения;
* работать с генетическими алгоритмами;
* решать обыкновенные дифференциальные уравнения и др.

# Извлечение информации из музыки

Каждый звуковой сигнал имеет множество характеристик, из которых следует отобрать нужные. Процесс получения информации для анализа называется извлечением объектов или извлечением сущностей (feature extraction). Такими характеристиками сигнала могут быть:

* частота пересечения нуля (zero crossing rate) – это частота изменения знака сигнала, т. е. частота, с которой сигнал меняется с положительного на отрицательный и обратно. Например, пусть есть некоторый сигнал и для него построен график амплитудной огибающей (рисунок 5) [11]

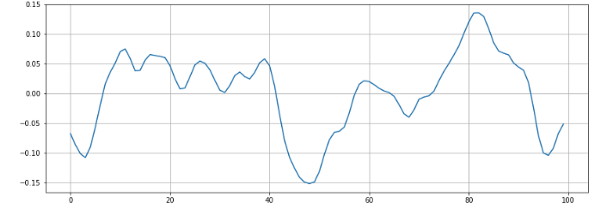


Рисунок 5 – График амплитудной огибающей сигнала

На графике видно 6 пересечений нуля. С помощью функции zero\_crossing библиотеки librosa проверим, так ли это. Код и результат представлены на рисунке 6:

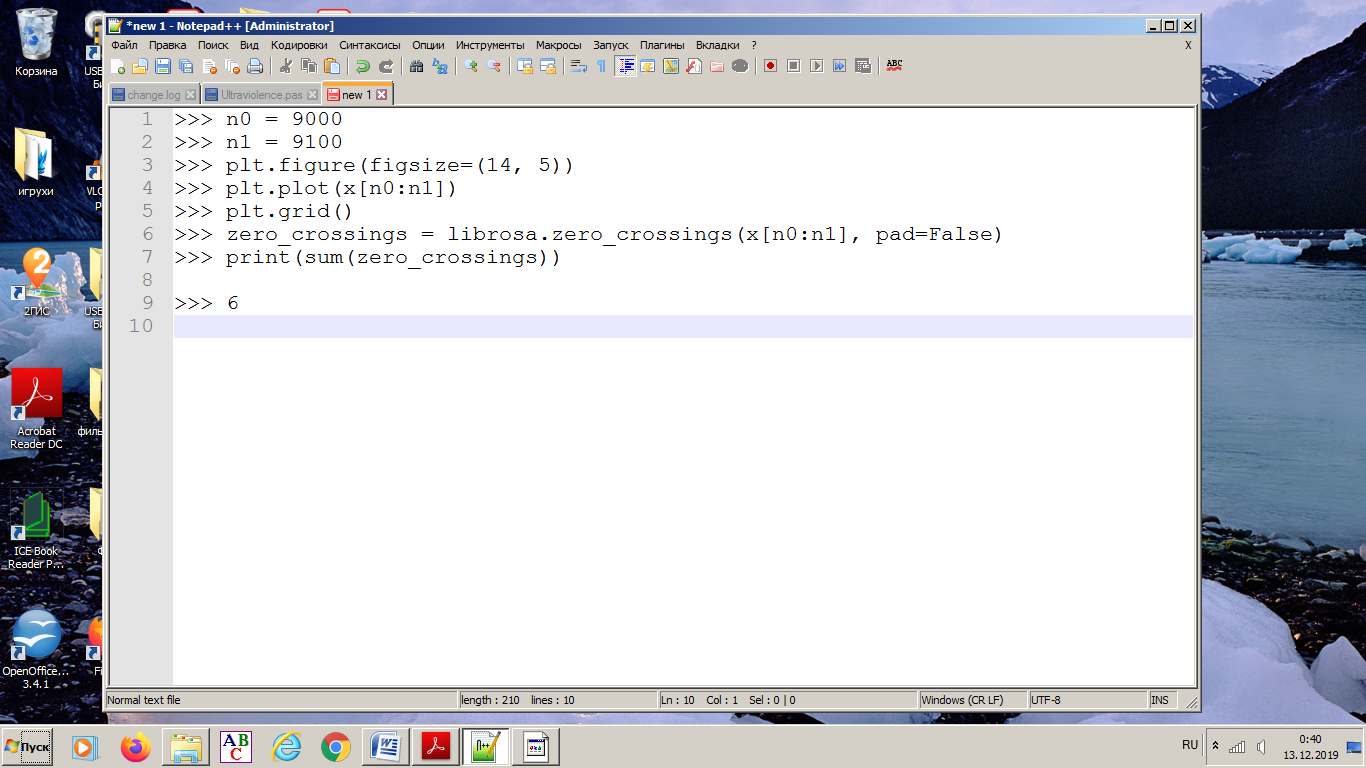


Рисунок 6 – Фрагмент программы, вычисляющей количество пересечений нуля амплитудной огибающей

Шесть переходов через нуль – всё верно. Функция zero\_crossing\_rate широко используется как для распознавания речи, так и для извлечения музыкальной информации. Для металла и рока этот параметр обычно выше, чем для других жанров, из-за большого количества ударных;

* спектральный центроид – указывает, где расположен «центр масс» звука и рассчитывается как средневзвешенное значение всех частот. В блюзовых композициях частоты равномерно распределены, и центроид лежит где-то в середине спектра. В металле же наблюдается выраженное смещение частот к концу композиции, поэтому и спектроид лежит ближе к концу спектра, как показано на рисунке 7:

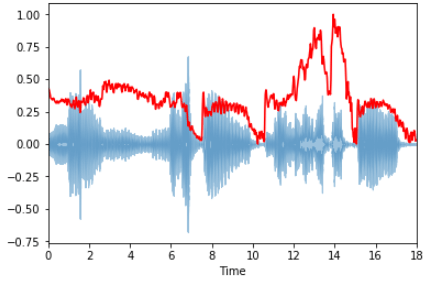


Рисунок 7 – График спектрального центроида

* мел-частотные кепстральные коэффициенты (MFCC) сигнала – небольшой набор характеристик (обычно около 10-20) которые сжато описывают общую форму спектральной огибающей. В основном этот параметр моделирует характеристики человеческого голоса. [10]

Далее с помощью функции пакета librosa – features.mfcc – рассчитываются мел-частотные кепстральные коэффициенты. В файле hmm\_trainer на основе полученных параметров и характеристик строится скрытая марковская модель. Параметр n\_components — определяет число скрытых состояний. Опираясь на исследования в области обучения с помощью СММ, было установлено, что относительно неплохие модели можно строить, используя 6-8 скрытых состояний, а для получения более сложных моделей можно использовать больше, около 20 состояний, но тогда и время обучения увеличится во много раз. Остальные параметры отвечают за сходимость EM-алгоритма, ограничивая число итераций, точность и определяя тип ковариационных параметров состояний. ЕМ-алгоритм, который также называют алгоритмом Баумы-Велша, используется для нахождения неизвестных параметров скрытой марковской модели. Он использует алгоритм прямого и обратного хода. Именно он и занимается обучением модели в файле trainholder.

Так как hmmlearn используется для обучения без учителя, то процесс обучения строится следующим образом: для каждого класса обучается своя модель, далее тестовый сигнал прогоняется через каждую модель, где по нему рассчитывается логарифмическая вероятность score каждой модели. В итоге класс, которому соответствует модель, выдавшая наибольшую вероятность, и является владельцем этого тестового сигнала. Реализация показана на рисунке 8:

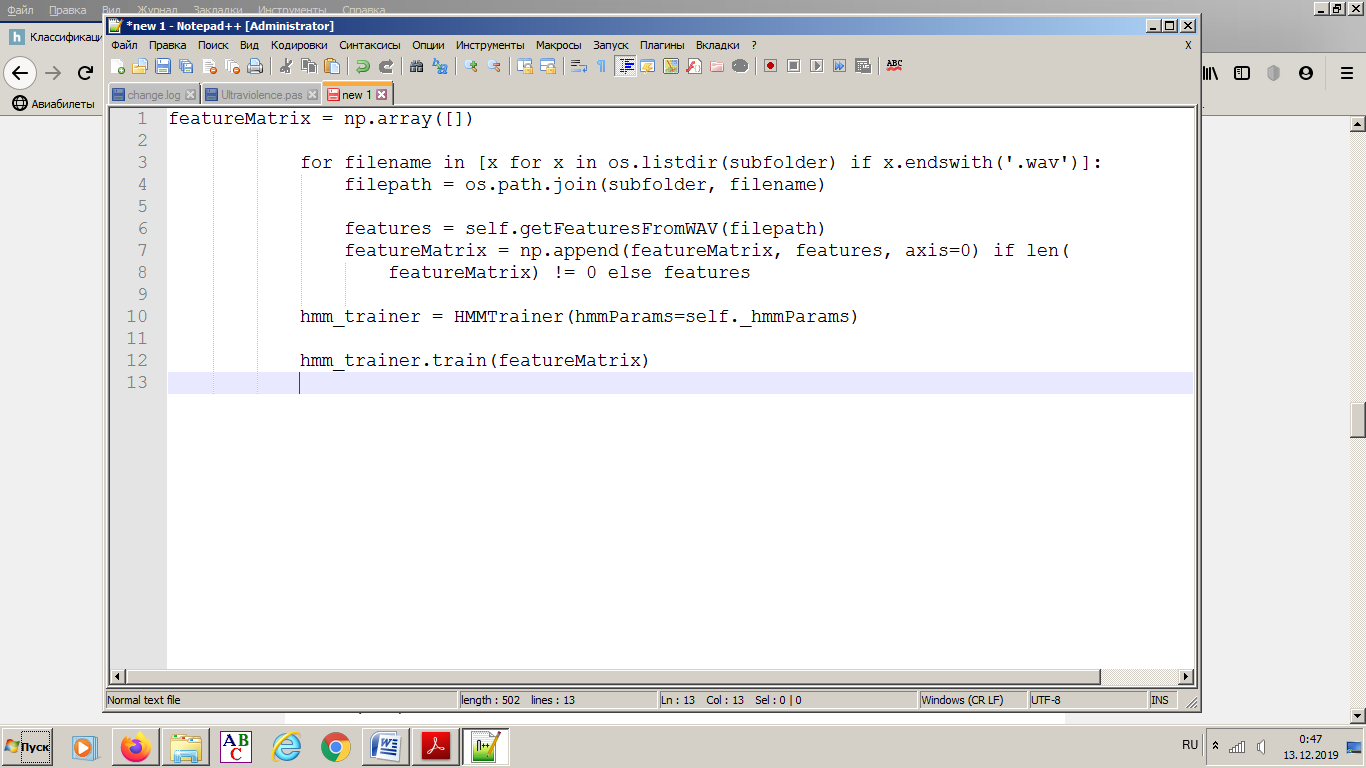


Рисунок 8 – Фрагмент реализации обучения модели

В общем случае, наш алгоритм классификации работает так: в папке с названием исполнителя для каждой из имеющихся в ней песен, то есть «.wav» файла, считаются MFCC, значения которых добавляются в матрицу признаков (np.array([])), причём строка соответствует фрейму, столбец соответствует номеру коэффициента из MFCC. После заполнения матрицы создается скрытая марковская модель для этого класса (исполнителя), и признаки передаются в EM-алгоритм для обучения. Далее осуществляется классификация: обращаемся к каждой модели и считаем для неё логарифмическую вероятность. Получаем сортированный по вероятностям набор классов, первый элемент которого показывает, кто является наиболее вероятным исполнителем данной песни.

Результат записывается в файл result.txt в программном файле test\_predict.py, в котором осуществляется классификация песен из предварительно созданной папки «…/Tests».

# Экспериментальное исследование обучающих данных

# Описание исходных данных

В обучающую выборку включены песни 6 исполнителей: Nirvana, Michael Jackson, Queen, 30 Seconds to Mars, Hurts, Green Day. Ниже представлен список песен каждой группы:

Данные выбирались из соображений, что некоторые песни указанных выше разных исполнителей можно считать похожими друг на друга, например, по звучанию на определённых отрезках песни, но при этом есть и совершенно разные по стилю композиции одного и того же исполнителя. Схожесть или же различия в звучании музыкальных композиций проще определить по их визуальному представлению, то есть по расположению и высоте частот на соответствующих им спектрограммах.

# Алгоритм решения задачи

Алгоритм решения поставленной задачи состоит из следующих шагов:

1. с помощью визуального анализа выделить скоррелированные признаки, схожесть и различие характеристик музыкальных композиций;
2. учитывая выбранные выше методы и характеристики, а также инструменты языка Python, разобраться в работе классификатора и обучить его на своих данных;
3. оценить точность работы классификатора;
4. указать возможные пути улучшения классификатора.

# Первичный визуальный анализ

Прежде чем работать с данными, необходимо как можно глубже понять их природу. Для этого обычно используется визуальный анализ. Зачастую данные визуально представимы в виде графиков. Для звуковых сигналов таким графиком является спектрограмма. Её несложно построить с помощью комплекта инструментов библиотек языка Python, написав небольшую программу. Из пакета scipy воспользуемся функцией scipy.io.wavfile.read (‘filename.wav’), которая позволяет извлечь из звукового файла отсчёты, представимые в виде одномерного массива (т. к. в нашем случае все «.wav» файлы конвертированы в одноканальном (моно) формате, если бы формат был стерео, то массив отсчётов был бы двумерным) и частоту, с которой они берутся. С помощью функций метода pyplot из библиотеки matplotlib можем построить спектрограмму аудиофайла, то есть график зависимости частоты от времени.

Например, рассмотрим первые 60 секунд композиции Another One Bites the Dust исполнителя Queen, изображённые в их визуальном представлении на рисунке 9.

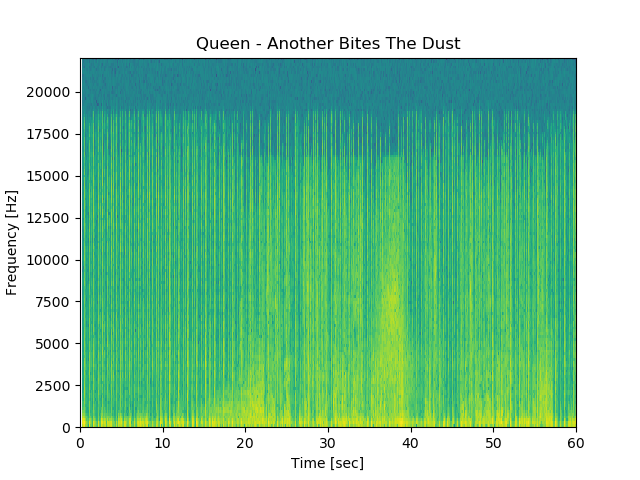


Рисунок 9 – Спектрограмма композиции Queen – Another One Bites the Dust

Теперь рассмотрим визуализацию первых 60 секунд композиции Billie Jean исполнителя Michael Jackson, представленную на рисунке 10:

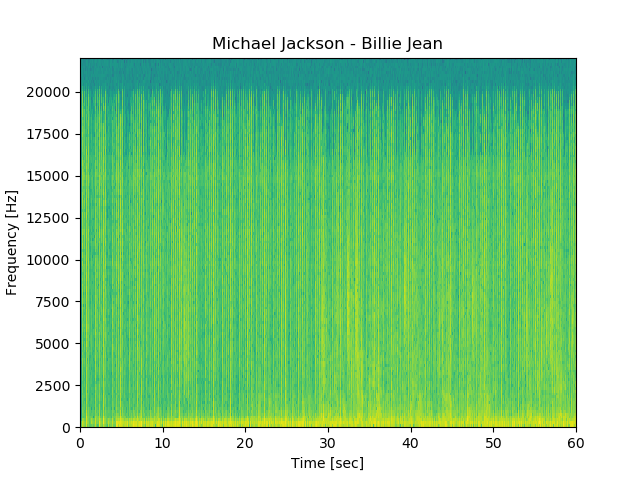


Рисунок 10 – Спектрограмма композиции Michael Jackson – Billie Jean

В их визуальном представлении этих композиций можно заметить некоторое сходство в расположении частот на временной шкале (Time, [sec]), в их высоте (Frequency, [Hz]) и интенсивности (чем светлее спектр, тем интенсивнее частоты).

Возьмём ещё пару спектрограмм для визуального анализа. Например, первые 60 секунд композиции Holiday исполнителя Green Day – спектрограмма рисунок 11:

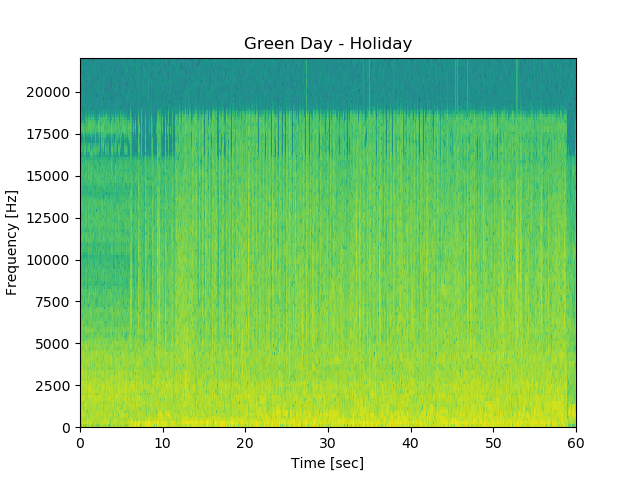


Рисунок 11 – Спектрограмма композиции Green Day – Holiday

Также рассмотрим визуальное представление композиции Come As You Are исполнителя Nirvana. Оно изображено в видео спектрограммы на рисунке 12.

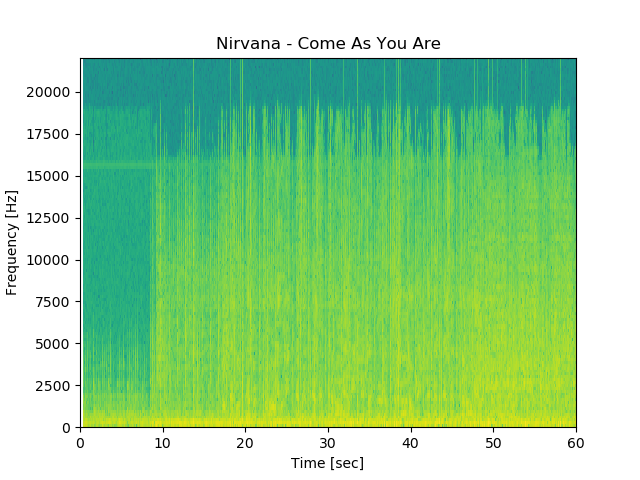


Рисунок 12 – Спектрограмма композиции Nirvana – Come As You Are

Из обоих графиков видно большое преобладание светлых оттенков спектра, то есть интенсивности частот, от десятой секунды до конца выбранного отрезка песни (60 секунд). Также на этом отрезке у обеих песен максимальная высота частот в среднем равна примерно 17750-17850 Hz. Таким образом можно сделать вывод о том, что рассмотренные пары композиций имеют схожие значения параметров. Соответственно, классификатор может отнести их к одному и тому же исполнителю, чего важно избежать. Подобный анализ данных помогает выявить общие черты в звучании композиций разных групп, чтобы отобрать те уникальные признаки каждой песни, на которые в дальнейшем можно ссылаться при составлении математической модели композиции и выборе методов классификации.

# Обучение классификатора

Обучающие данные для классификатора подготовлены вручную и имеют структуру небольшого каталога таким образом: для каждого исполнителя создаётся папка с его именем в качестве названия, содержащая несколько его песен в формате «.wav». Затем эти папки помещаются в родительскую папку. Причём в ней должны быть только подпапки-исполнители. При изучении построений классификаторов, описанных в различных статьях и книгах о машинном обучении, было выяснено, что при обучении на основе алгоритма скрытых марковских моделей, когда для каждого исполнителя, который представляется классом, будет создаваться своя модель, эти модели рекомендуется сохранять в отдельную папку. Каждая из полученных моделей сохраняется в виде файла с разрешением «.pkl» и названием – именем класса.

В качестве среды Python для обучения классификатора и его дальнейших тестов выбран облачный сервис для обучения нейронных сетей Google Colaboratory, описанный в пункте 2.1. Предпринимались попытки обучить классификатор в интерпретаторе PyCharm, но сделать этого не удалось из-за несовместимости версий некоторых библиотек с версиями пакетов, предназначенных специально для этого интерпретатора в операционной системе Windows. В Google Colaboratory имеются практически все нужные пакеты, к тому же можно легко установить недостающий модуль командой !pip install <module\_name>. Данные для обучения были сохранены в папке folder на Google диске, который был подключён к созданному блокноту. Модели хранятся в папке Saves.

Для лучшего понимания структуры классификатора, написанного на Python, и процесса его обучения были изучены и протестированы уже готовые классификаторы. Исходя из этого было принято решение для удобства создать файл config.py, в котором указываются пути к папкам с обучающими данными, сохранёнными моделями, с тестовыми данными, а также определяются некоторые параметры. Результаты тестов записываются в текстовый файл «result.txt» При изучении работы классификаторов с использованием скрытых марковских моделей как метода обучения на основе фактов из статей об обучении классификаторов, было выяснено, что для лучших результатов следует строить 10-12 скрытых состояний, а для получения более сложных моделей – 20. Но при этом отмечалось, что время обучения может увеличиться во много раз, а достаточно хорошие результаты можно получить, строя и 6-8 скрытых состояний. В файле config.py параметр n\_components=6 определяет, что число скрытых состояний для марковской модели будет равно 6.

На рисунках 13 и 14 представлены скриншоты процесса обучения классификатора песен по их исполнителям.

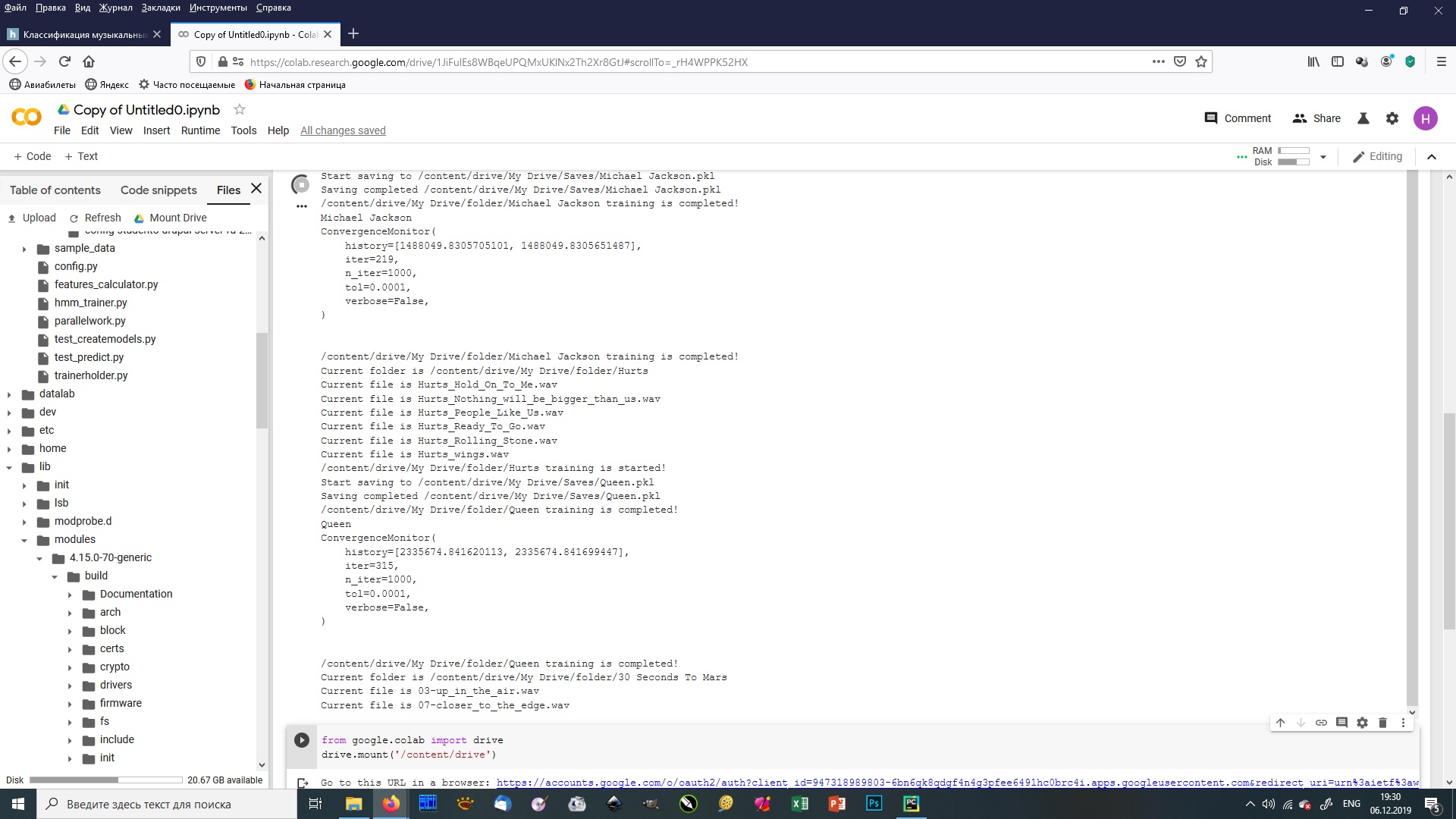


Рисунок 13– Процесс обучения классификатора

Для удобства в процессе обучения на экран выводится название текущего файла, из которого извлекаются параметры и строится модель, а также имя класса-папки (исполнителя), в котором он находится. Классификатору поступают классы на обучение в том же порядке, в котором они расположены в родительской папке.

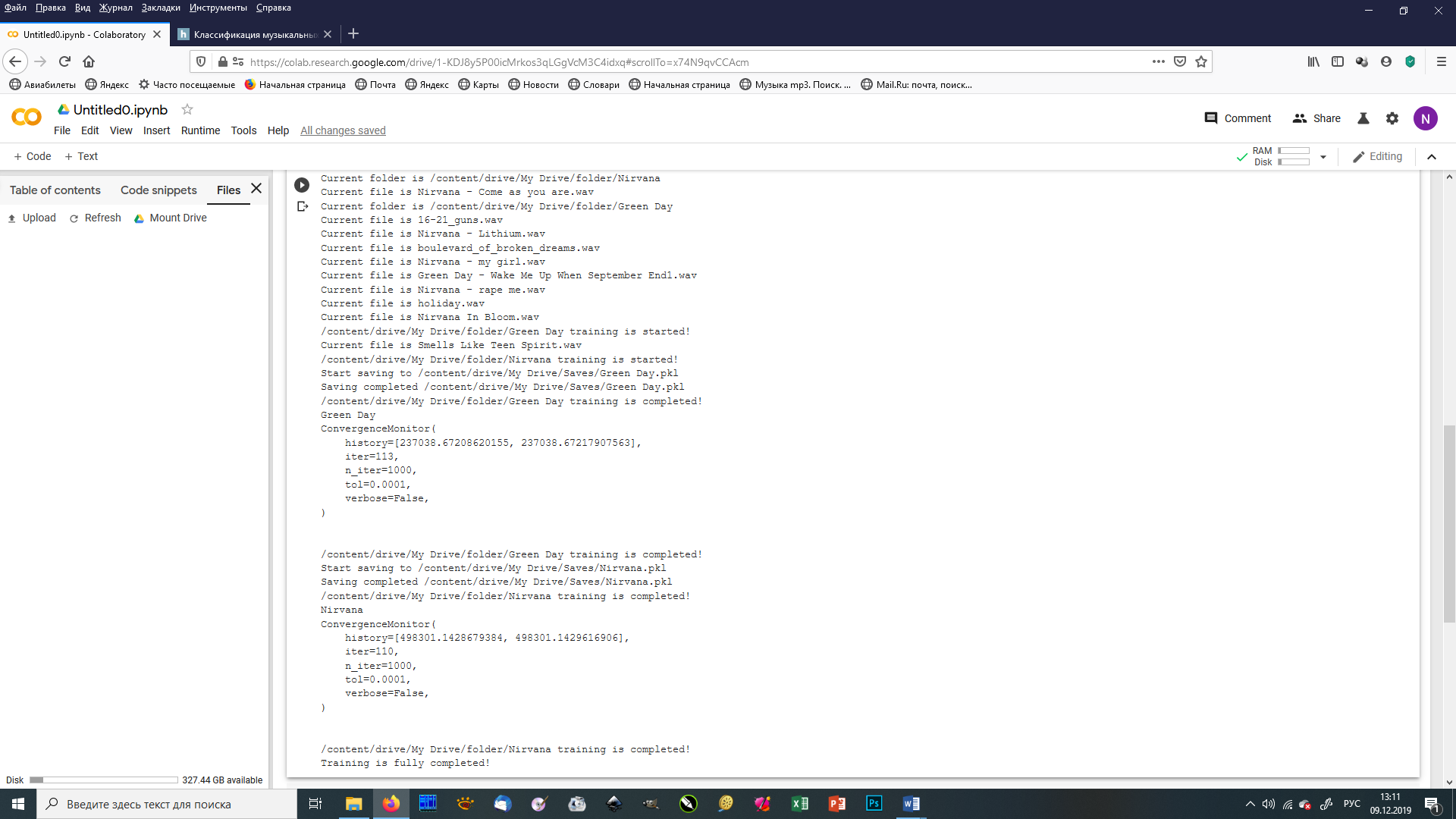


Рисунок 14 – Окончание обучения классификатора

На рисунках 13 и 14 «Current file is <file\_name>» – файл, для которого создаётся модель в текущий момент времени. Как только обучение и сохранения моделей в отдельном каталоге закончилось, классификатор выводит на экран надпись «<Class\_name> training is completed». Также выводится на экран количество итераций EM-алгоритма – iter, точность и тип ковариационных параметров состояний.

# Анализ полученных результатов

В качестве тестовых данных в папке Tests были выбраны композиции, которые использовались в обучении классификатора и те, которых не было в обучающих данных (в скобках указано имя исполнителя) (рисунок 15)



Рисунок 15 – Тестовые композиции

По окончанию тестирования классификаторы были получены следующие результаты (рисунок 16):

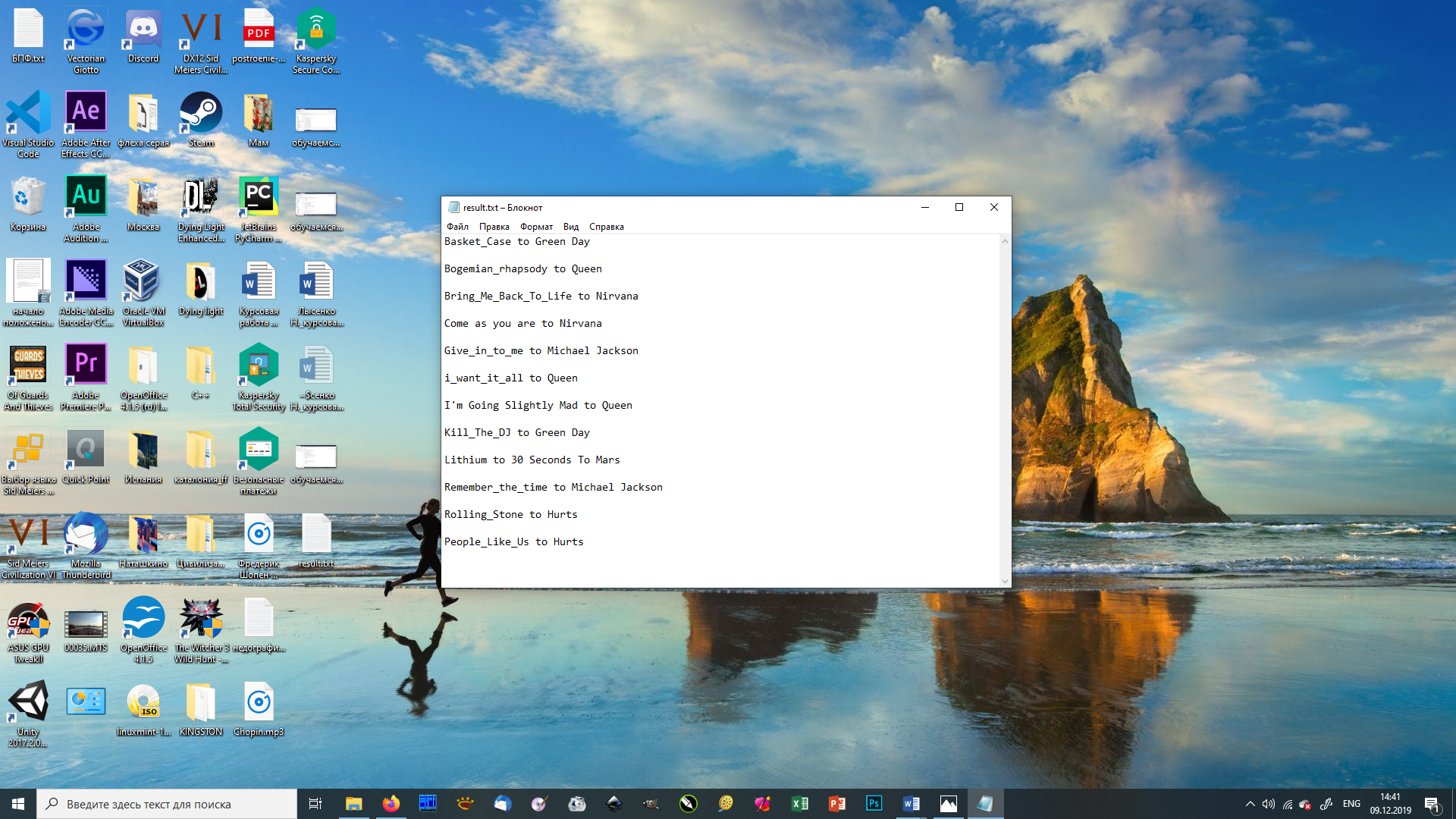


Рисунок 16 – Результаты теста

Классификатор неверно определил исполнителей только двух композиций похожих стилей: Bring\_Me\_Back\_To\_Life to Nirvana (на самом деле исполнитель – 30 Seconds to Mars) и Lithium to 30 Seconds To Mars (на самом деле исполнитель – Nirvana). Если учитывать то, что композиций для обучения было взято небольшое количество – 40 (для лучших результатов следует брать больше 100), и при обучении количество скрытых состояний 6, а не 16, то есть модели получаются сравнительно простые (но при этом такого числа состояний достаточно для получения весьма неплохих результатов), то 2 ошибки из 12 песен, в целом, хороший результат.

Для улучшения классификатора можно попробовать извлекать больше информации из музыкального файла, например, помимо мел-кепстральных коэффициентов и количества точек в преобразовании Фурье получать ещё частоту пересечения нуля и центральный спектроид, описанные в пункте 2.2. Также можно указывать около 20 скрытых состояний в моделях Маркова для получения более интересных моделей и поставлять классификатору достаточно большое количество композиций в качестве данных для обучения.

# **ЗАКЛЮЧЕНИЕ**

В процессе выполнения работы был изучен язык программирования Python и его инструменты для реализации алгоритмов классификации на основе скрытых марковских моделей. Также получены знания о представлении звуковых сигналов в цифровом виде, их обработке с помощью средств Python, возможные пути получения характеристик звуковых сигналов, необходимых для их анализа. Кроме того, при изучении данной темы была разобрана и отлажена работа классификатора музыкальных композиций по исполнителям и произведена его настройка под выбранные для его обучения данные.

В будущем планируется создать собственный классификатор на основе знаний, полученных в процессе написания данной работы, но классифицировать он будет музыкальные композиции уже не по исполнителям, а по жанрам. Для этого из музыкального файла нужно будет извлекать в разы больше информации и строить на её основе более интересные модели для обучения. Задача идентификации музыкальных композиций является весьма сложной, потому что часто не существует общепринятых классификационных признаков того или иного жанра. Также сложность её состоит в том, что даже человек допускает ошибки в определении исполнителя или жанра композиции, а он является примером идеального идентификатора на данных момент.

В связи с ростом числа музыкальных интернет-сервисов и их постоянной заинтересованностью улучшить рекомендации для своих пользователей, подобная задача классификации музыкальных композиций по жанрам является весьма актуальной в настоящее время.

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Коэльо Л.П., Ричарт В. Построение систем машинного обучения на языке Python. 2-е издание/ пер. с англ. Слинкин А.А. – М.: ДМК Пресс, 2016. – 302 с.: ил. ISBN 978-5-97060-330-7
2. Программирование на Python том I, 4-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 992 с., ил. ISBN 978-5-93286-210-0
3. Солдатова О.П. Использование многослойного персептрона для распознавания жанров музыкальных композиций // Известия Самарского научного центра Российской академии наук, т. 18, № 4(4), 2016
4. Скрытая Марковская Модель [Электронный ресурс]. URL:https://ru.wikipedia.org/wiki/Скрытая\_марковская\_модель (Дата обращения – 10.11.2019)
5. Мел-кепстральные коэффициенты и распознавание речи [Электронный ресурс]. URL: https://habr.com/ru/post/140828/ (Дата обращения – 10.11.2019)
6. Алгоритм Баума-Велша [Электронный ресурс]. − URL: https://ru.wikipedia.org/wiki/Алгоритм\_Баума\_—\_Велша (Дата обращения – 15.11.2019)
7. Практическое применение преобразования Фурье для анализа сигналов [Электронный ресурс]. − URL: https://habr.com/ru/post/269991/ (Дата обращения – 22.10.2019)
8. Вероятностная интерпретация классический моделей машинного обучения [Электронный ресурс]. − URL: https://m.habr.com/ru/post/343800/ (Дата обращения – 25.11.2019)
9. Классификация музыкальных композиций по исполнителям с помощью скрытых марковских моделей [Электронный ресурс]. − URL: https://m.habr.com/ru/post/351462/ (Дата обращения – 29.11.2019)
10. Жанровая классификация на Python [Электронный ресурс]. – URL: https://proglib.io/p/python-music-classifier (Дата обращения – 01.12.2019)

# **ПРИЛОЖЕНИЕ**

Листинг программы:

Config.py

# file contains parameters for test\_createmodels.py and test\_predict.py

# 0

# classifier params

n\_components = 6

cov\_type = 'diag'

n\_iter = 1000

# MFCC params

nmfcc = 40

nfft = 1024

##TRAIN##

# define folder which contains songs for training or subfolders of different classes with songs for training

train\_folder = "/content/drive/My Drive/folder"

# folder for saving models

save\_folder = "/content/drive/My Drive/Saves/"

# debug mode for training

debug\_train = True

# max number of threads for training

max\_threads = 2

##TEST##

# folder which containst testing songs

test\_folder = "../Tests"

# folder which contains saved models

models\_folder = save\_folder

# debug mode

debug\_mode = True

# output scores for test songs under every model

table\_score = False

# file for logs. if it is None, no file will be created

result\_file = "result.txt"

features\_calculator.py

import librosa

from librosa.feature import mfcc

import sklearn.preprocessing

# 1

class FeauturesCalculator:

    """

    Opens file, calculates features and returns them

    Attributes

    ----------

    \_res\_type : str

        parameter for librosa.load

        defines resample type

    \_nfft : int

    \_nmfcc : int

    \_hop\_length : int

        info for calculating mfcc (for librosa.feature.mfcc)

    \_scale : bool

        scale mfcc or not

    """

    def \_\_init\_\_(self, nfft, nmfcc, hop\_length=512, res\_type='scipy', scale=True):

        self.\_res\_type = res\_type

        self.\_nfft = nfft

        self.\_nmfcc = nmfcc

        self.\_hop\_length = hop\_length

        self.\_scale = scale

    def getFeaturesFromWAV(self, filename):

        audio, sampling\_freq = librosa.load(

            filename, sr=None, res\_type=self.\_res\_type)

        features = librosa.feature.mfcc(

            audio, sampling\_freq, n\_mfcc=self.\_nmfcc, n\_fft=self.\_nfft, hop\_length=self.\_hop\_length)

        if self.\_scale:

            features = sklearn.preprocessing.scale(features)

        return features.T

hmm\_trainer.py

import os

from hmmlearn import hmm

import joblib

import numpy as np

import warnings

warnings.simplefilter('ignore', DeprecationWarning)

# 2

class HMMParams:

    """

    Defines significant params for HMM

    """

    def \_\_init\_\_(self, n\_components=4, cov\_type='diag', n\_iter=1000, tol=1e-4):

        self.n\_components = n\_components

        self.cov\_type = cov\_type

        self.n\_iter = n\_iter

        self.tol = tol

class HMMTrainer:

    """

    Wrapper for GaussianHMM

    """

    def \_\_init\_\_(self, hmmParams=HMMParams()):

        self.\_hmm = hmm.GaussianHMM(n\_components=hmmParams.n\_components,

                                    covariance\_type=hmmParams.cov\_type, n\_iter=hmmParams.n\_iter, tol=hmmParams.tol)

    def train(self, X):

        np.seterr(all='ignore')

        self.\_hmm.fit(X)

    def get\_score(self, input\_data):

        return self.\_hmm.score(input\_data)

    def get\_score\_samples(self, input\_data):

        return self.\_hmm.score\_samples(input\_data)

    def get\_monitorInfo(self):

        return self.\_hmm.monitor\_

    def save(self, folder\_name, class\_name, debug\_mode=False):

        """

        folder\_name : str

            folder which contains output

        class\_name : str

            name of output file

        debug\_mode : bool

            prints debug info is true

        """

        if folder\_name[-1] != '/':

            folder\_name += '/'

        filename = folder\_name + class\_name + '.pkl'

        if debug\_mode:

            print('Start saving to ' + filename)

        joblib.dump(self.\_hmm, filename)

        if debug\_mode:

            print('Saving completed ' + filename)

    def load(self, folder\_name, class\_name, debug\_mode=False):

        """

        folder\_name : str

            folder which contains output

        class\_name : str

            name of output file

        debug\_mode : bool

            prints debug info is true

        """

        for filename in [x for x in os.listdir(folder\_name) if (x.endswith('.pkl') and (class\_name in x))]:

            filepath = os.path.join(folder\_name, filename)

            if debug\_mode:

                print('Start loading  ' + filename)

            self.\_hmm = joblib.load(filepath)

            if debug\_mode:

                print('Loading completed ' + filename)

trainerholder.py

import os

import numpy as np

from scipy.io import wavfile

from hmm\_trainer import \*

from features\_calculator import FeauturesCalculator

# 3

class TrainHolder:

    """

    TrainHolder creates, fits, saves, loads HMM

    Attributes

    ----------

    \_hmmParams : HMMParams

        Parameters of Gaussian HMM

    \_mfccCalculator : FeaturesCalculator

        Opens '.wav' file and calculates mfcc-features

    \_models : List of HmmTraines

        Contains HMMs

    save\_folder : str

        Path for saving models during training

        If  is None

            models are not saving during training

    """

    def \_\_init\_\_(self, n\_components=4, cov\_type='diag', n\_iter=1000, nfft=512, nmfcc=20, save\_folder=None):

        self.\_hmmParams = HMMParams(n\_components, cov\_type, n\_iter)

        self.\_mfccCalculator = FeauturesCalculator(nfft=nfft, nmfcc=nmfcc)

        self.\_models = []

        self.save\_folder = save\_folder

    def train(self, dataFolder, shuffle=True, seed=None, debug\_mode=False):

        """

        dataFolder : str

            path to folder which contains songs for training or

             subfolders with songs for training

            the name of folder/subfolder refers to the name of class

        shuffle : bool

            shuffle mfcc features  of different songs before fitting or not

            shuffling usually increases robustness of model, but also reduces training speed

        seed : int

            seed for random shuffling, useless if shuffle is False

        debug\_mode : bool

            prints debug info is true

        """

        np.random.seed(seed)

        if debug\_mode:

            print('Current folder is ' + dataFolder)

        lstdirs = [os.path.join(dataFolder, dirname) for dirname in os.listdir(

            dataFolder) if os.path.isdir(os.path.join(dataFolder, dirname))]

        noSubFolders = not lstdirs

        if noSubFolders:

            lstdirs.append(dataFolder)

        for subfolder in lstdirs:

            if not os.path.isdir(subfolder):

                continue

            label = subfolder[subfolder.rfind('/') + 1:]

            featureMatrix = np.array([])

            for filename in [x for x in os.listdir(subfolder) if x.endswith('.wav')]:

                filepath = os.path.join(subfolder, filename)

                if debug\_mode:

                    print('Current file is ' + filename)

                features = self.getFeaturesFromWAV(filepath)

                featureMatrix = np.append(featureMatrix, features, axis=0) if len(

                    featureMatrix) != 0 else features

            if debug\_mode:

                print(subfolder + ' training is started!')

            hmm\_trainer = HMMTrainer(hmmParams=self.\_hmmParams)

            np.random.shuffle(featureMatrix)

            hmm\_trainer.train(featureMatrix)

            self.\_models.append((hmm\_trainer, label))

            if self.save\_folder is not None:

                hmm\_trainer.save(self.save\_folder, label, debug\_mode)

            if debug\_mode:

                print(subfolder + ' training is completed!')

                self.\_debConvergeInfo(hmm\_trainer, label)

        if noSubFolders and debug\_mode:

            print(dataFolder + ' training is completed!')

    def test(self, folder\_name, result\_file=None, table\_score=False, debug\_mode=False):

        """

        folder\_name : str

            path to folder which contains songs for predict

        result\_file : str

            path to file for logs. if is None logs are printing to cmd

        table\_score : bool

            prints scores of every model on current song

        debug\_mode : bool

            prints debug info is true

        """

        if debug\_mode:

            print('Tests are started!')

        resultStr = ""

        for input\_file in [x for x in os.listdir(folder\_name) if x.endswith('.wav')]:

            if debug\_mode:

                print('Current file is ' + input\_file)

            filepath = os.path.join(folder\_name, input\_file)

            resultStr += self.testFile(filepath, table\_score=table\_score)

        if debug\_mode:

            print('Concluded!\n Printing results...\n')

        # if result\_file = None => print to cmd

        if result\_file is None:

            print(resultStr)

        else:

            with open(result\_file, 'w') as f:

                f.write(resultStr)

        if debug\_mode:

            print('Results are printed!')

    def testFile(self, filepath, result\_file=None, table\_score=False):

        """

        filepath : str

            path to song for predict

        result\_file : str

            path to file for logs. if is None logs are printing to cmd

        table\_score : bool

            prints scores of every model on current song

        returns string with logs

        """

        res = ""

        features = self.getFeaturesFromWAV(filepath)

        scores = {}

        for hmm\_model, label in self.\_models:

            score = hmm\_model.get\_score(features)

            scores[label] = score

            if table\_score:

                res += '    Score ' + str(score) + ' for ' + label + '\n'

        similarity = sorted(scores.items(), key=lambda t: t[1], reverse=True)

        i = 0

        for item in similarity:

            i += 1

            res += str(i) + ' ' + item[0] + '\n'

        # Print the output

        res += os.path.basename(filepath) + ' to ' + \

            str(similarity[0][0]) + '\n'

        if result\_file is None:

            print(res)

        else:

            with open(result\_file, 'w') as f:

                f.write(res)

        return res

    def save(self, folder\_name, debug\_mode=False):

        for model, label in self.\_models:

            model.save(folder\_name, label, debug\_mode)

    def load(self, folder\_name, class\_names, debug\_mode=False):

        for label in class\_names:

            model = HMMTrainer()

            model.load(folder\_name, label, debug\_mode)

            self.\_models.append((model, label))

    def getFeaturesFromWAV(self, filename):

        return self.\_mfccCalculator.getFeaturesFromWAV(filename)

    def \_debConvergeInfo(self, model, label):

        print(label)

        print(model.get\_monitorInfo())

        print('\n')

parallelwork.py

import threading

import os

import queue

import time

from trainerholder import TrainHolder

# 4

def run(queueLock, workQueue, trainh, exitFlag, shuffle, seed, debug\_mode):

    while not exitFlag[0]:

        queueLock.acquire()

        if not workQueue.empty():

            folder\_name = workQueue.get()

            queueLock.release()

            trainh.train(folder\_name, shuffle, seed, debug\_mode)

            workQueue.task\_done()

        else:

            queueLock.release()

            time.sleep(1)

def parallel\_train(trainh: TrainHolder, folder\_name=None, folder\_list=None, max\_threads=None, shuffle=True, seed=None, debug\_mode=False):

    """

    folder\_name : str

    folder\_list : str

        find all classes in folder\_name and/or one by one in folder\_list

    max\_threads : int

        max number of threads for training

        each thread trains one class

    debug\_mode : bool

        prints debug info

    """

    fn, fl = folder\_name is None, folder\_list is None

    if fn and fl:

        print('No folder specified')

        return

    path\_list = []

    if not fn:

        for dirname in os.listdir(folder\_name):

            subfolder = os.path.join(folder\_name, dirname)

            if not os.path.isdir(subfolder):

                continue

            path\_list.append(subfolder)

    if not fl:

        for subfolder in folder\_list:

            if not os.path.isdir(subfolder):

                continue

            path\_list.append(subfolder)

    num\_threads = len(path\_list) if max\_threads is None else min(

        max\_threads, len(path\_list))

    queueLock = threading.Lock()

    workQueue = queue.Queue()

    exitFlag = [False]

    print('Threads Creating')

    threadlist = [threading.Thread(target=run, args=(queueLock, workQueue, trainh, exitFlag, shuffle, seed, debug\_mode))

                  for i in range(num\_threads)]

    for var in threadlist:

        #var.daemon = True

        var.start()

    print('Threads started')

    print(len(path\_list))

    queueLock.acquire()

    for folder\_name in path\_list:

        workQueue.put(folder\_name)

    queueLock.release()

    workQueue.join()

    exitFlag[0] = True

    for t in threadlist:

        t.join()

    print('Training is fully completed!')

test\_createmodels.py

from trainerholder import TrainHolder

from parallelwork import parallel\_train

import config

# define folder which contains songs for training or subfolders of different classes with songs for training

train\_folder = config.train\_folder

# 5

# folder for saving model

save\_folder = config.save\_folder

# create classificator

clf = TrainHolder(n\_components=config.n\_components, cov\_type=config.cov\_type,

                  n\_iter=config.n\_iter, nfft=config.nfft, nmfcc=config.nmfcc)

# define folder for saving models during training

clf.save\_folder = save\_folder

# create threads for different classes

parallel\_train(clf, train\_folder, debug\_mode=config.debug\_train,

               max\_threads=config.max\_threads)

test\_predict.py

from trainerholder import TrainHolder

import config

class TestPredict:

    """

    Sample describes how to use TrainHolder

    Parameters

    ----------

    models\_folder : str

        path to folder which contains saved models

    class\_names : list of str

        names of classes. saved model files must contain these names

    """

    def \_\_init\_\_(self, models\_folder, class\_names):

        self.clf = TrainHolder(n\_components=config.n\_components, cov\_type=config.cov\_type,

                               n\_iter=config.n\_iter, nfft=config.nfft, nmfcc=config.nmfcc)

        self.clf.load(models\_folder, class\_names, config.debug\_mode)

    def predictFolder(self, test\_folder):

        self.clf.test(test\_folder, table\_score=config.table\_score,

                      debug\_mode=config.debug\_mode, result\_file=config.result\_file)

    def predictFile(self, test\_file):

        self.clf.testFile(test\_file, table\_score=config.table\_score,

                          result\_file=config.result\_file)

    def checkConverges(self):

        self.clf.whoConverged()

if \_\_name\_\_ == "\_\_main\_\_":

    # folder which contains trained models

    models\_folder = config.models\_folder

    # folder which contains test songs

    test\_folder = config.test\_folder

    # define here which classes should be loaded

    # name of file with model should contain name of corresponding class

    class\_names = ['30 Seconds To Mars', 'Nirvana', 'Green Day',

                   'Hurts', 'Michael Jackson', 'Queen']

    predictor = TestPredict(models\_folder, class\_names)

    predictor.predictFolder(test\_folder)