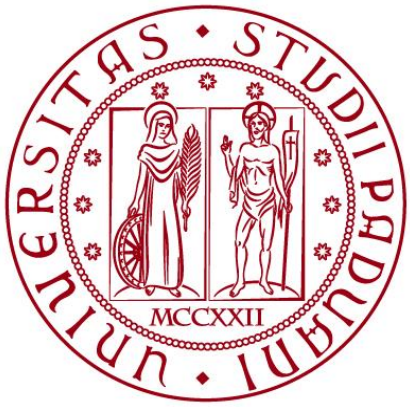


UNIVERSITÀ DEGLI STUDI DI

PADOVA

Dipartimento di Ingegneria dell'Informazione

TESI DI LAUREA TRIENNALE IN
INGEGNERIA INFORMATICA

ELABORAZIONE DI UN DATASET DI PARTITURE MUSICALI PER SISTEMI DI CREATIVITA' COMPUTAZIONALE

Relatore: Prof. Antonio Roda'
Co-relatore: Dott. Filippo Carnovalini

23/09/2019

Laureando: Davide Lionetti

ANNO ACCADEMICO 2018 / 2019

Sommario

Il lavoro qui affrontato tratta l'elaborazione e la conversione del *Nottingham Dataset*, raccolta di brani tradizionali di origine irlandese, per facilitare il suo utilizzo in applicazioni di creatività computazionale. La conversione del corpus fornisce una versione di questo in formato MusicXML, in origine fornito in formato ABC, inserito all'interno di un database costruito appositamente. Come strumento di conversione è stata utilizzata la libreria per Python "Music21", che ha permesso inoltre l'estrazione dei metadati necessari per il popolamento del database. Nella seconda parte è stato implementato un sistema di generazione di melodie tramite l'utilizzo delle catene di Markov, estraendo dal Nottingham dataset i valori di probabilità necessari per l'utilizzo di questo processo. Al termine di questo lavoro sono presentati alcuni spartiti contenenti melodie monofoniche, generati seguendo un processo di creatività computazionale al seguito della elaborazione del Nottingham Dataset.

Indice

Introduzione

La creatività computazionale studia le metodologie attraverso cui tecnologie informatiche possono emulare la creatività umana, puntando alla costruzione di un programma o di un algoritmo capace di, per esempio, poter comporre una melodia inedita. I progetti di creatività computazionale mirano alla produzione di opere che gli osservatori umani possano considerare creative, generando artificialmente idee e processi innovativi. Questi processi sono spesso applicabili a svariati campi come l'arte, la letteratura, la cucina, l'architettura, l'ingegneria e la musica.
La creatività computazionale applicata all'ambito musicale che tratterò in questo lavoro si focalizza sulla creazione di nuovi componimenti, generati tramite l'utilizzo delle Catene Di Markov che introdurrò nel secondo capitolo.

Definiamo prima di tutto cos'è la creatività: la creatività può essere definita come l'abilità umana di creare idee innovative e sorprendenti, a volte del tutto sconosciute in precedenza. Sono state identificate tre tipologie di creatività umana; la creazione di idee inedite può essere ottenuta tramite processi di *combinazione*, *esplorazione* o *trasformazione*. Ciascuna tipologia dà il nome a uno specifico processo creativo, come descritto in [1].
Per illudere l'osservatore, la creatività computazionale si basa sui processi che governano la prima di queste tre tipologie, ovvero la *creatività combinatoria* (combinational creativity): produrre nuove idee partendo da combinazioni sconosciute di idee creative già esistenti. Questo può essere fatto creando associazioni tra idee in precedenza collegate indirettamente. L'analogia è una forma di creatività combinatoria: mette in luce il rapporto che la mente coglie fra due o più soggetti che hanno, nella loro costituzione, qualche tratto comune.

La *creatività esplorativa* si basa sull'esplorazione di alcune forme di pensiero accettate culturalmente: spazi concettuali che definiscono uno stile (e.g. uno specifico stile di pittura o scrittura). Lo spazio è definito e limitato da un insieme di regole generative, solitamente queste sono del tutto implicite. Ogni struttura prodotta seguendo le regole stilistiche si adatterà allo stile in questione, proprio come ogni frase scritta seguendo la sintassi italiana seguirà le regole grammaticali imposte dalla lingua. Il processo di creatività esplorativa consiste nell'esplorare un certo ambiente stilistico per ricercare alcune sfumature non ancora notate o scoperte prima. Questa esplorazione può portare a scoprire nuovi potenziali, o nuovi limiti dello spazio in questione. La presenza di questi limiti è l'elemento fondante per il passaggio da questo processo al processo successivo: *la creatività trasformativa*. Il processo di creatività trasformativa nasce dal superamento di certi limiti imposti

dallo spazio stilistico che portano all'alterazione di uno o più regole che lo definiscono. Così facendo si è in grado di creare un nuovo ambiente stilistico, quindi di andare a generare qualcosa di totalmente nuovo. Più il vincolo cambiato è stilisticamente rilevante per la definizione dello spazio, più sorprendenti saranno le nuove idee che potranno nascere. A volte potrebbe portare ad un cambio così radicale da richiedere molti anni per essere accettato dalla mentalità collettiva. Un esempio lampante potrebbe essere l'innovazione pittorica portata da Vincent Van Gogh riconosciutagli solo dopo la sua morte, avendo venduto un solo dipinto durante la sua vita.

In questo testo ci soffermeremo sulla prima di queste tre tipologie: la creazione per combinazione.
Nel campo della creatività computazionale qui presa in considerazione, degli esempi di utilizzo della creatività combinatoria sono le combinazioni di pattern armonici o ritmici, assimilati dalla macchina nella fase di analisi, con cui il calcolatore sarà in grado di comporre brani inediti.

Lo scopo principale di questo lavoro è di facilitare l'utilizzo e l'elaborazione del *Nottingham music database*, una raccolta di spartiti di musica popolare, tramite la sua conversione dal formato *ABC* al più diffuso e moderno formato *MusicXML*: l'obiettivo di questo lavoro punta a facilitarne l'utilizzo per algoritmi di generazione melodica o in altri lavori di ricerca correlati. Oltre alla traduzione per facilitarne il reperimento ci occuperemo anche della riorganizzazione del corpus in un database in cui ad ogni brano verranno accostati il titolo, la tonalità e la metrica.
Il Nottingham music database è un noto corpus contenente più di 1200 melodie folk inglesi e Statunitensi, già utilizzato per ricerche nel capo del *machine learning* e della creatività computazionale per la generazione di melodie. Questa collezione di componimenti folkloristici, generalmente utilizzati per accompagnare tradizionali balli popolari, fu trascritta e assemblata per la prima volta da Eric Foxley, il creatore di questa raccolta (come descritto in [2]). La start-up Jukedeck che lavora nell'ambito della composizione algoritmica, servendosi di algoritmi di intelligenza artificiale ha fornito tramite il sito web Git-hub una versione ripulita del Nottingham dataset già in precedenza disponibile in formato ABC. Inoltre, Jukedex ha fornito, tramite la piattaforma Soundcluod, un esempio di composizione effettuata da un modello allenato con il Nottingham dataset, che può essere ascoltato online[1].
L'obiettivo secondario di questa tesi di ricerca è progettare un algoritmo di creazione di melodie utilizzando un processo stocastico: per l'esattezza utilizzando le catene di Markov. L'algoritmo estrarrà i valori delle probabilità di transizione, necessarie per il lavoro di generazione musicale, dal

[1] https://soundcloud.com/jukedeck/nottingham-dataset-simple-composition

Nottingham Music Dataset, per poi comporre brani musicali nello stesso stile. Il modello statistico, usando il calcolo delle probabilità, studia le sequenze di associazioni armoniche, ovvero l'alternanza e la durata delle note, presenti in un insieme di opere di partenza. Questo processo individua le relazioni esistenti tra ogni elemento melodico presente nei brani, per poterle ripetere in modo randomico in fase di composizione, su misura delle precedenti opere che ha analizzato. I processi stocastici per l'emulazione della creatività umana sono introdotti all'inizio del secondo capitolo.

Confrontiamo ora i due formati ABC e MusicXML per comprendere l'importanza della conversione effettuata.
MusicXML è un sistema di codifica XML studiato per rappresentare il sistema di notazione musicale occidentale, è un formato libero, rilasciato sotto licenza GNU GPL, senza diritti d'autore che può essere utilizzato gratuitamente. MusicXML fu progettato per istaurare un linguaggio comune tra le applicazioni di creazione e visualizzazione di spartiti musicali; grazie alla sua vasta diffusione questo linguaggio è ormai supportato dalla maggioranza dei programmi di notazione musicale. Possiamo quindi affermare che il formato MusicXML è diventato il nuovo standard open-source per la condivisione e la scrittura di spartiti interattivi. Oggi più di 240 applicazioni supportano MusicXML, tra cui uno dei più diffusi programmi per la scrittura di spartiti: *Musescore.*
Musescore è uno dei programmi open-source di notazione musicale più utilizzato per la scrittura e la lettura di spartiti, ottima alternativa ai programmi commerciali come Finale e Sibelius. In questo lavoro utilizzerò il programma Musescore per visualizzare gli spartiti risultanti della conversione del Nottingham dataset, e per mostrare i risultati di generazione di musica inedita ottenuti applicando le catene di Markov. [3]

Il protocollo ABC definisce un linguaggio per la scrittura di componimenti musicali in formato di testo (ASCII). Utilizza la rappresentazione anglosassone della scala cromatica che identifica le 7 note principali come delle lettere in sequenza: parte dal La indicato dalla lettera A fino ad arrivare al Sol indicato dalla lettera G. Le note sono associate a dei particolari simboli di durata, tonica e tonalità più alcuni metadati relativi al brano come il compositore, il titolo e il metro. Questa notazione fu originalmente studiata per tenere traccia dei componimenti tradizionali proveniente da paesi anglofoni come l'Inghilterra la Scozia e l'Irlanda. Nonostante questo linguaggio sia gradualmente andato in disuso, vari programmi forniscono l'estensione per supportarlo in cui generalmente si ottiene la conversione dei file abc in file MIDI. A prova di questo vi è la motivazione data dalla start-up Jukedeck per il suo lavoro di modifica e pulizia del Nottingham-dataset, atto a facilitare la conversione del corpus in formato MIDI.

Qui sta la grande limitazione dell'utilizzo del linguaggio ABC per scopi accademici e di ricerca: in questi ambiti risulta più utile poter convertire ogni brano di interesse in notazione di sparito musicale piuttosto che in formato MIDI. Per questo in ambito accademico è stato scelto come standard la notazione in MusicXML facilmente convertibile in spartito con l'utilizzo di famosi programmi come Musescore. Per questo motivo abbiamo ritenuto utile la conversione dell'intero *Nottingham music database* in formato MusicXML. [4]

Per il lavoro di conversione e reperimento dei metadati quale titolo, tonalità e metro di ogni componimento ho utilizzato Music21: una libreria per Python che fornisce numerosi strumenti per l'analisi e la creazione di file musicali come per esempio spartiti scritti in MusicXML o in ABC. Il lavoro di conversione ottenuto tramite l'utilizzo di Music21 verrà introdotto nel capitolo 1. [5]

Per la creazione del database ho utilizzato MySQL, un RDBMS (Relational Database Management System) open source disponibile gratuitamente che utilizza il linguaggio SQL (Structured Query Language). Per ora implementato solo a livello locale si intende rendere disponibile online il Nottingham music database tradotto alla fine di tutto il lavoro. [6]

NOTE: L'inizio dei capitoli, dell'introduzione e la bibliografia devono stare sempre su pagine dispari.

1. Elaborazione del Nottingham Music Dataset

1.1 Music21

Music 21 è una libreria Python open-source per gli studi di musicologia assistiti da un calcolatore. Questa libreria permette l'utilizzo di numerosi strumenti per la scrittura, l'elaborazione o la conversione di grandi insiemi di dati nel campo della notazione musicale. Music21 è stata creata nel 2008 dal MIT (Massachussets etc) (in collaborazione con numerose università come Harvard, Smith e Mount Holyoke).
Music21 è in continuo aggiornamento ed espansione grazie soprattutto ai numerosi contribuenti esterni.

Con l'utilizzo di Music21 sono stato in grado di poter affrontare il lavoro di conversione e manipolazione del Nottingham Dataset. Ho ottenuto l'intero corpus di opere dalla pagina web GitHub, fornite dalla start-up Jukedeck nel formato ABC, modificato e ripulito per facilitare la sua conversione in formato MIDI. Ho utilizzato "Visual Studio Code" come editor di testo, il linguaggio di programmazione Python (terza versione) e il programma Musescore per visualizzare il lavoro di conversione sotto forma di spartito musicale. Oltre alla libreria Music21 ho utilizzato anche il modulo 'os' di python che mi è servito per estrarre i percorsi dei file in formato ABC durante la conversione e per ottenere i nomi con cui i file sono stati salvati nel Nottingham dataset.
Come prima cosa, dopo aver installato Music21, ho dovuto modificare alcuni elementi di configurazione che mi hanno permesso di poter specificare Musescore come programma di lettura dei file in formato MusicXML. Per fare questo ho utilizzato il modulo Environment di Music21, creando un'istanza di environment e utilizzando il metodo UserSettings() che mi ha permesso di fare delle modifiche di configurazione. Music21 definisce un insieme di chiavi di configurazione che permettono di indicare le preferenze d'interazione della libreria con il proprio computer. Per andare a definire Musescore come applicazione predefinita per aprire i file MusiXML ho utilizzato la chiave 'musicxmlPath'. Grazie a questa chiave di configurazione, ogni qual volta vorrò mostrare un oggetto di tipo Score utilizzando Musescore, basterà utilizzare il metodo score.show().Di seguito le prime righe dello script denominato "AbcToMusicXmlConverter.py"

```
from music21 import *
import mysql.connector
import os




#Utlizzando il metodo Usersetting() posso impostare le mie preferenze creand
o un oggetto di environment configuration
us = environment.UserSettings()
#con la chiave 'musicxmlPath' vado ad impostare MuseScore come lettore prede
finito dei file Musicxml
us['musicxmlPath']='C:/Program Files/MuseScore/bin/MuseScore3.exe'
```

Nel prossimo paragrafo riporto le parti principali del mio lavoro di conversione del Nottingham Music dataset. [7]

1.2 Conversione dei file ABC in MusicXML

Per la conversione dei file ABC in formato MusicXML mi sono servito del modulo 'converter' di music21. Questo modulo permette di importare file di vari formati compreso ABC, e di convertirli in oggetti *stream.Score* . Ottenuto un oggetto di tipo score potrà essere mostrato con il metodo *score.show()*(tramite Musescore*)* o salvato nel formato predefinito (MusicXML) con il metodo score.write(). Utilizzando il metodo converter.parse('filepath') e inserendo come parametro il percorso del file che si vuole importare, si otterrà un oggetto di tipo Stream. La classe Stream è una componente fondamentale della libreria Music21, definita nella documentazione come un "contenitore di elementi musicali indicizzati", a cui appartengono gli oggetti di tipo Score(spartito). Il Nottingham dataset è composto da 16 raccolte di brani e ogni raccolta è interpretata da Music21 come un oggetto *stream.Opus*, una sottoclasse della classe stream che identifica tutti i file contenenti una raccolta di brani, come nel nostro caso. Per ottenere tutti i singoli spartiti all'interno dell'oggetto opus mi sono servito del metodo opus.scores. Da ogni singolo brano ho estratto i principali attributi di interesse da poter inserire nel database, per facilitare la manipolazione del Nottingham dataset. I dati estratti sono: il titolo, un id interno ad ogni singolo opus per indicizzarne i brani, la tonalità e il metro. Essendo brani tradizionali, l'autore è andato perduto e non è presente nei file ABC. Music21 interpreta il titolo e l'id interno come oggetti di tipo "*metadati".* Per ottenere questi due attributi ho utilizzato i metodi metadata.title e metadata.number della classe stream.metadata. Per la tonalità del

brano ho utilizzato il metodo stream.analyze(‘parola-chiave’) inserendo come parola chiave ‘key’. Per il metro mi sono servito dell’oggetto TimeSignature, appartenente alla classe stream.meter.
Poiché la classe meter è una sottoclasse di stream e non di score per applicare il metodo meter.TimeSignature al mio oggetto, devo ricercare tale metodo tra tutte le classi di stream utilizzando il metodo recourse().getElementsByClass(meter.TimeSigature)[0].

Mostriamo un esempio di conversione del terzo brano del Nottingham dataset intitolato “Barry’s Favourite”, mostriamo lo spartito risultante utilizzando il metodo score.show() servendoci di Musescore.

```
abcPath1= "C:/Users/david/Documents/Il mio mondo/Corsi Università/
Tesi Triennale/nottingham-dataset-master/ABC_cleaned/ashover.abc"

opus= converter.parse(abcPath1)

for el in opus.scores:
    el.show()
    #estraggo il metro del brano
    metro= el.recurse().getElementsByClass(meter.TimeSignature)[0]

    k = el.analyze('key')# ottengo la tonalità del componimento

    title= el.metadata.title #ottengo il titolo

    internalId= el.metadata.number# ottengo l'internalId
```

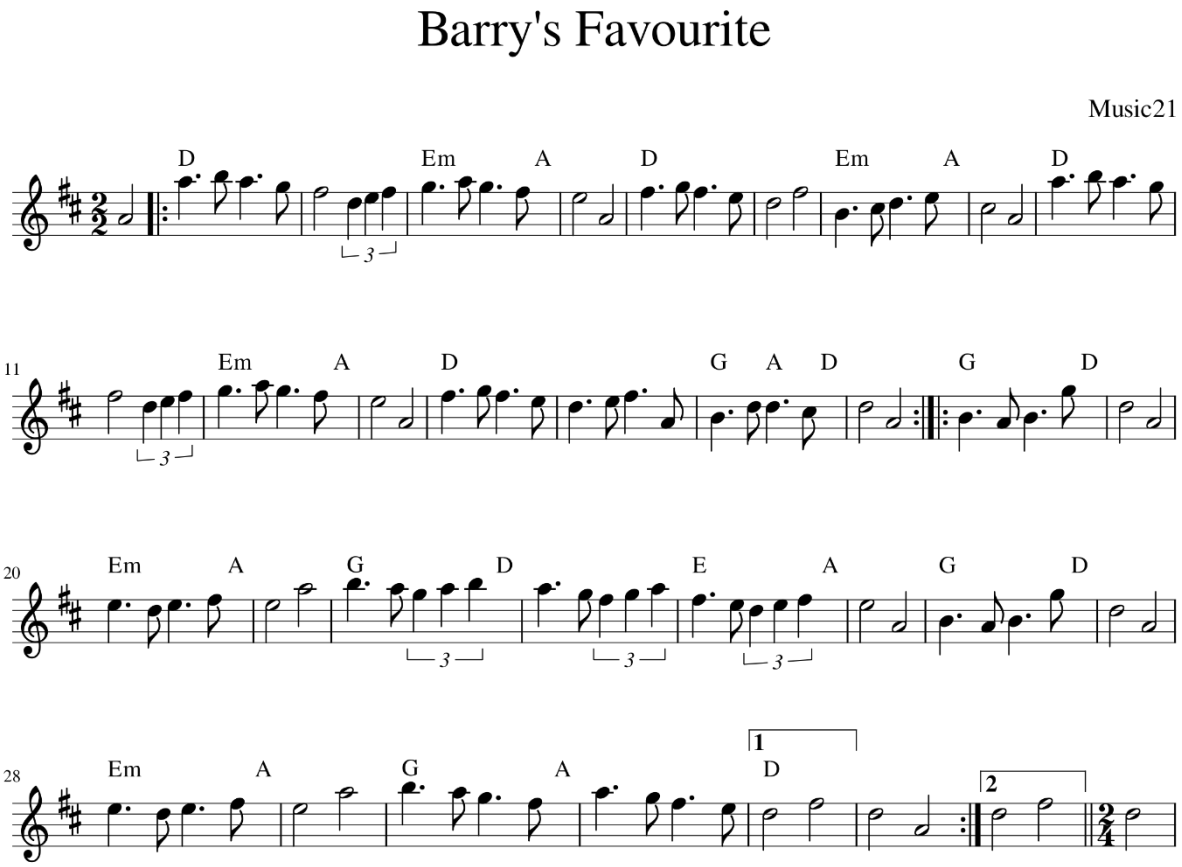


Figura 1.1: Spartito del terzo brano del Nottingham dataset, ottenuto tramite l'utilizzo di Musescore.

Per poter ottenere tutti i brani dai 16 file che compongono il dataset ho dovuto ottenere il percorso di ogni singolo file ABC da inserire poi nel metodo converter.parse(). Per far questo mi sono servito di alcuni metodi del modulo 'os' di python. Per ottenere la lista di tutti i nomi dei file del dataset ho creato una cartella nominata ABC_files nella cartella corrente dove ho messo tutti i file. Il metodo os.listdir('./ABC_files) ritorna una lista dei nomi dei file contenuta nella cartella che ho specificato utilizzando il percorso relativo "./ABC_files".

```
#lista dei nomi dei file nella directory ABC_files
abcFiles = os.listdir('./ABC_files')

for abcFile in abcFiles:
  abcFilePath = os.path.abspath(os.path.join('ABC_files', abcFile))
  #ottengo il percorso assoluto di ogni file

  opus = converter.parse(abcFilePath)

  for score in opus.scores:
    #estraggo il metro del brano
    metro= score.recurse().getElementsByClass(meter.TimeSignature)[0]

    k = score.analyze('key')# ottengo la tonalità del componimento

    title= score.metadata.title #ottengo il titolo

    internalId= score.metadata.number# ottengo l'internalId
```

Per ogni file nel ciclo for ottengo il percorso assoluto della directory corrente utilizzando il metodo os.path.abspath(). Poiché i file si trovano nella cartella "ABC_files" (della directory corrente), utilizzo il metodo join per unire al percorso assoluto la cartella ABC_files: os.path.abspath(os.path.join(ABC_files, abcFile)).

Ho salvato ogni singolo brano convertito in MusicXML con il metodo score.write(fp='percorso desiderato'): grazie all'argomento "fp" posso specificare il percorso dove voglio salvare i file. Nella directory corrente è presente una cartella XML_files al fine di contenerli tutti; per ottenere il percorso assoluto della directory corrente ho utilizzato il metodo os.getcwd(). Tramite il metodo join ho ottenuto percorso assoluto della cartella "XML_files", salvandolo in una variabile nominata "directory_Write". Per far sì che ogni file score venga salvato come il titolo del brano cui si riferisce ho utilizzato il metodo join per unire al percorso salvato in diretcoryWrite il titolo del brano (ottenuto con il metodo score.metadata.title).

Di seguito il codice completo utilizzato per la conversione di ogni brano:

```
from music21 import *
import mysql.connector
import os



#Utlizzando il metodo Usersetting() posso impostare le mie preferenze creand
o un oggetto di environment configuration
us = environment.UserSettings()
#con la chiave 'musicxmlPath' vado ad impostare MuseScore come lettore prede
finito dei file Musicxml
us['musicxmlPath']='C:/Program Files/MuseScore/bin/MuseScore3.exe'


'''
Utilizzo la funzione 'parse' del modulo converter per convertire i file ABC
in oggetti di music21, testando la coversione con il metedo .show()
che come impostato in precedenza con il modulo enviroment.UserSetting() apre
 il programma MuseScore e mostra gli spartiti tradotti
'''

#istauro una connessione con il server specificando come database
'nottingham'
mydb = mysql.connector.connect(
  host="localhost",
  user="Davide",
  passwd="********",
  database="nottingham")

mycursor = mydb.cursor() # permette di interrogare il database

#currentDir salva il percorso della directory corrente
currentDir = os.getcwd()

#aggiunge a currentDir la cartella XML_file dove salviamo i file convertiti
in Musicxml
directoryWrite = os.path.join(currentDir, 'XML_files')

abcFiles = os.listdir('./ABC_files') #lista dei nomi dei file
nella directory ABC_files

for abcFile in abcFiles:
  #ottengo il percorso assoluto di ogni file
  abcFilePath = os.path.abspath(os.path.join('ABC_files', abcFile))
```

```
opus = converter.parse(abcFilePath)

  for score in opus.scores:
    #estraggo il metro del brano
    metro= score.recurse().getElementsByClass(meter.TimeSignature)[0]

    k = score.analyze('key')# ottengo la tonalità del componimento
    title= score.metadata.title #ottengo il titolo
    internalId= score.metadata.number# ottengo l'internalId

    abspath=os.path.join(directoryWrite, score.metadata.title)
    #aggiunge al percorso della directoryWrite il titolo del brano

    score.write(fp=abspath)
    #Salvo la conversione del brano nella cartella dedicata:'XML_files'

    score.show() #mostro lo spartito tramite Musescore
```

1.3 Creazione e popolamento del database

1.3.1 Creazione

Per facilitare l'utilizzo e l'elaborazione del Nottingham dataset, in fase di elaborazione mi è sembrato utile poter raccogliere tutti i brani in un unico database in cui si forniscono inoltre le informazioni correlate ad ogni brano. Per svolgere questo compito mi sono servito di uno dei più famosi "Relational *Database Management System*", MySQL che mi ha permesso di creare un database locale nel mio computer con l'obbiettivo di caricarlo in rete alla fine di tutto il lavoro. Per semplicità di utilizzo, ho implementato solo una tabella chiamata "nottingham" che racchiude tutti i brani con i relativi attributi più il nome del file da cui il brano proviene. Oltre a distinguere i brani per titolo ho anche deciso di tenere traccia della divisione del corpus in 16 file differenti, aggiungendo un attributo "originalFileName". Dopo aver istallato MySQL e avviato il programma server all'interno del *MySQL Workbench*, è stato necessario importare nel mio file python il modulo mysql.connector per poter iniziare il lavoro. Grazie al modulo mysql.connector è possibile creare, modificare e visualizzare il proprio database all'interno del rispettivo file Python, rivelandosi un'ottima forma di comunicazione con il Database Management System. Inoltre è possibile eseguire delle *query*(richieste) utilizzando il linguaggio SQL grazie al metodo connector.execute("Query SQL").
Per l'implementazione del database ho utilizzato un file a parte nominato *"NottinghamDatabase.py"*, dentro il quale troviamo l'istaurazione della connessione con il database di MySQL in cui ho inserito le credenziali richieste durante l'istallazione:

```
import mysql.connector

#instauro la connessione con il mio database MySQL

mydb = mysql.connector.connect(
  host="localhost",
  user="Davide",
  passwd="*********",
)
```

Successivamente ho ottenuto un oggetto del tipo MySQLcursor con il metodo mydb.cursor(), che mi permette di effettuare delle interrogazioni al database tramite il linguaggio SQL. Il primo comando SQL è stato "CREATE DATABASE" per poi accederci direttamente con il metodo connector.connect() specificando al suo interno il nome del database, in questo caso "nottingham".

```
mycursor = mydb.cursor()

#creo un database
mycursor.execute("CREATE DATABASE nottingham")

#accedo al database appena creato
mydb = mysql.connector.connect(
  host="localhost",
  user="Davide",
  passwd="**********",
  database="nottingham")
```

Durante la creazione della tabella chiamata "*corpus"* ho valutato positivamente l'idea di inserire come chiave primaria un Id di tipo intero che si auto incrementa ogni qual volta si inserisce un nuovo record nella tabella. In questo modo posso tenere traccia del numero di tutte le opere inserite al suo interno. Nel prossimo script mostro la creazione della tabella "corpus", formata da sei colonne: un id di tipo intero auto incrementante, il titolo del brano, la tonalità detta "*k*", il metro nominato "*meter*", *l'originalId* (intende l'indice interno ad ogni raccolta di brani) e il nome della raccolta da cui proviene il brano detto "idriginalFileName".
Infine, per analizzare il contenuto della tabella dopo l'inserimento dei dati, ho interrogato il database utilizzando il linguaggio SQL: SELECT * FROM corpus. Per ottenere tutti i record ottenuti dall'operazione di selezione ho utilizzato il metodo cursor.fetchall() per poi stamparli con un semplice ciclo for.

```
#creo la tabella corpus
#imposto come chiave primaria un intero che si auto icrementa ad ogni
record inserito
mycursor.execute("CREATE TABLE  corpus (id INT AUTO_INCREMENT PRIMARY KEY, t
itle VARCHAR(255), k VARCHAR(255), meter VARCHAR(255),originalId INT, origin
alFileName VARCHAR(255)) ")


mycursor.execute("SELECT * FROM corpus")

myresult = mycursor.fetchall() #Usiamo il metodo fetchall() per recupera tut
te i record dell'istruzione di selezione.

for x in myresult:
  print(x)
```

1.3.2 Popolamento

Il popolamento del database è stato effettuato nello stesso script in cui ho convertito i file. Come prima cosa ho istaurato una connessione al server specificando il database “nottingham” (creato in precedenza) e istanziato l’oggetto connection.cursor.

```
#istauro una connessione con il server specificando come database
il 'nottingham'
mydb = mysql.connector.connect(
  host="localhost",
  user="Davide",
  passwd="**********",
  database="nottingham")

mycursor = mydb.cursor() # permette di interrogare il database
```

Per inserire i dati nel database ho utilizzato il comando SQL: INSERT INTO corpus. Quando si effettuano delle modifiche ad una tabella, MySQL richiede di utilizzare il metodo connection.commit() per applicare l’aggiornamento. In fase di inserimento ho dovuto trasformare alcuni dati estratti dai brani in semplici stringhe poiché risultavano degli oggetti esterni non gestibili da MySQL (oggetti definiti all’interno di Music21). Con il metodo score.analyze(‘key’) si ottiene un oggetto key.Key che ho trasformato in una stringa utilizzando il metodo k.tonicPitchNameWithCase.

Il metodo per l'estrazione del metro ritorna un oggetto del tipo meter.TimeSignature, il problema è stato risolto utilizzando il metodo metro.ratioString riuscendo ad ottenere il metro del brano come una semplice stringa. Di seguito il codice completo:

```
opus = converter.parse(abcFilePath)

  for score in opus.scores:
    #estraggo il metro del brano
    metro= score.recurse().getElementsByClass(meter.TimeSignature)[0]

    k = score.analyze('key')# ottengo la tonalità del componimento
    title= score.metadata.title #ottengo il titolo
    internalId= score.metadata.number# ottengo l'internalId

    abspath=os.path.join(directoryWrite, score.metadata.title)
    #aggiungendo al percorso della directoryWrite il titolo del brano

    score.write(fp=abspath)# Salvo la conversione del brano nella
    cartella dedicata 'XML_files'

    score.show() #mostro lo spartito tramite Musescore

    #per inserire un elemento nella tabella usa la chiave primaria
    INSERT INTO

    sql = "INSERT INTO corpus (title,k,meter,originalId,originalFileName)
           VALUES (%s,%s,%s,%s,%s)"

    val = (title,k.tonicPitchNameWithCase, metro.ratioString, internalId
       ,abcFile)

    mycursor.execute(sql, val)


    mydb.commit()  #utilizziamo commit() per applicare il cambiamento
    alla tabella
```

2. Sviluppo di un sistema di generazione di melodie

2.1 Algoritmi per la composizione musicale

La composizione musicale algoritmica comprende un insieme di tecniche per la composizione di nuove melodie attraverso la progettazione e l'utilizzo di un algoritmo. La composizione algoritmica potrebbe essere descritta come una sequenza di regole (istruzioni, operazioni) per la risoluzione di un particolare problema di combinazioni di elementi musicali, in un numero finito di passaggi. Il concetto di composizione algoritmica non è qualcosa di nuovo, Pitagora infatti, (intorno al 500 a.C.) credeva che la matematica e la musica non fossero studi separati.
Possiamo identificare diversi algoritmi di composizione musicale introdotti in questi ultimi decenni, secondo la specifica tecnica che utilizzano per ricreare il processo creativo.
Una composizione algoritmica può essere riassunta in due passaggi:

1. jSi formalizza un procedimento compositivo: in questo processo si stabiliscono tutte le operazioni necessarie per comporre un nuovo brano su cui ci si vuole soffermare. In questa fase si deve scegliere uno specifico approccio che definisca la tecnica applicata per la generazione finale.
2. Si esegue l'algoritmo creato che lavorerà solo sui parametri scelti nel punto uno.

Esistono tanti approcci possibili per la composizione algoritmica, presentiamo i tre principali:

1. *Modelli Matematici*: utilizzano il caso e/o l'improvvisazione per emulare la creatività.
   Processi aleatori composti da una serie di Procedimenti Stocastici (solitamente catene di Markov): si dice stocastico un procedimento atto a descrivere situazioni che variano in base a leggi probabilistiche (e non deterministiche). Questi processi estraggono i valori di probabilità desiderata dall'analisi di numerosi componimenti, generalmente tutti di uno stesso genere, che utilizzerà per generare il brano inedito. Grazie alla loro bassa complessità computazionale i processi stocastici sono ideali per le applicazioni in tempo reale. Per questo sono stati utilizzati ampiamente in passato per applicazioni di composizione algoritmica (e.g., M e Jam Factory, guardare Zicarelli, 1987).

2. *Grammatiche formali*: Processo che definisce un sistema di regole per delineare matematicamente un insieme (di solito infinito) di sequenze finite di simboli appartenenti ad un alfabeto: come tutte

le possibili successioni di note o pause in uno spartito musicale prese dall'alfabeto delle notazioni musicali. Il tutto volto a generare i parametri per definire le possibili fasi di una composizione.

3. *Sistemi basati sulla conoscenza:* Sono anch'essi sistemi basati su grammatiche formali con la capacità aggiuntiva di poter ridefinire le grammatiche durante le inizializzazioni successive dell'algoritmo, sfruttando processi di intelligenza artificiale. Vi sarà una prima fase di allenamento della rete neurale, attraverso l'utilizzo di un training-set di brani musicali ben definito. Terminata la prima fase, l'algoritmo avrà ottenuto le regole grammaticali che lo condizioneranno nelle scelte successive. In questo tipo di algoritmi sono molto utilizzati i "*Constraith graph*", grafi di vincoli: si possono imporre all'algoritmo dei vincoli basati sulla teoria musicale per rendere più realistico e piacevole il lavoro di composizione; tale tecnica nell'ambito dell'intelligenza artificiale viene chiamata *"Constraint optimization Problem"*, problema di soluzione di vincoli. Un algoritmo di intelligenza artificiale, seguendo i vari vincoli teorici imposti e avendo definito un obiettivo da raggiungere definito da una funzione obiettivo, creerà un albero di possibili soluzioni scegliendo il cammino che lo porterà all'obiettivo con meno sforzo computazionale possibile.

Ci sono molte altre strategie per emulare la creatività umana tramite l'utilizzo di un algoritmo. I programmi basati su un solo modello algoritmico raramente riescono ad ottenere risultati esteticamente soddisfacenti. Per questo è sempre più frequente utilizzare delle soluzioni "ibride", in cui si utilizzano simultaneamente più algoritmi diversi. L'unico problema principale dei sistemi ibridi è la loro crescente complessità computazionale e la necessità di molte risorse per combinare e testare gli algoritmi.

In questo lavoro ci soffermeremo sul primo approccio che riguarda l'utilizzo di processi stocastici, definiti utilizzando catene di Markov. Si definisce processo stocastico markoviano (o di Markov), un processo aleatorio in cui la probabilità di transizione, che determina il passaggio di stato del sistema, dipende solo dallo stato del sistema immediatamente precedente (proprietà di Markov) e non da come si è giunti a questo stato. Si dice processo non markoviano un processo aleatorio per cui non vale la proprietà di Markov.

Il modello statistico, usando il calcolo delle probabilità, studia le sequenze di associazioni armoniche, ovvero l'alternanza e la durata delle note di cui è composto un insieme di brani di partenza: individua le relazioni esistenti tra ogni elemento melodico per poterle ripetere in modo randomico in fase di composizione, su misura delle precedenti opere che ha analizzato. [8]

2.2 Catene di Markov

Un processo stocastico viene utilizzato per descrivere una sequenza di eventi casuali che si susseguono nel tempo, quindi descritti da una variabile temporale che li indicizza.
In questo lavoro utilizzeremo un processo stocastico per scegliere una sequenza adeguata di note tali da comporre un brano assimilabile come una creazione umana. Essendo legata alla variabile di tempo una catena di Markov può essere tempo-continua o tempo-discreta, a seconda dello spazio di definizione della variabile di tempo. Un processo di Markov è formato da uno *spazio degli stati* detto "S", che comprende l'insieme degli eventi che lo compongono, ed uno *spazio dei parametri.* In un processo di Markov, ad ogni occorrenza della variabile temporale 't' è associata una variabile aleatoria indipendente 'X' definita nello spazio degli stati. Il cambio di stato del sistema viene detto *transizione di stato* e la probabilità associata a questo evento viene detta *probabilità di transizione.*
Definiamo un processo di Markov *omogeneo,* ovvero un processo stocastico che gode della proprietà di Markov, in cui il valore della variabile aleatoria dello stato futuro X_{t+1} dipende solo dal valore della variabile aleatoria al tempo corrente X_t.
Quando un processo di Markov è omogeneo e costituito da un insieme di stati finiti (numerabile), si parlerà di catena *stocastica o catena di Markov.* Una catena di Markov descrive un sistema che si trova in uno specifico stato ad ogni occorrenza della variabile temporale 't.' Nella applicazione presentata in questo testo lo spazio degli stati della catena di Markov è composto dall'insieme delle dodici note della scala armonica (A, A#, B, C…G, G#).
In sintesi una catena di Markov omogenea è una sequenza di variabili aleatorie indipendenti X_1, X_2, X_3,… che godono della proprietà di Markov, definite nello spazio degli stati e indicizzate dalla variabile di tempo.
Definiamo la probabilità di transizione di stato come la probabilità di ottenere il valore $X_{t+1}=j$ preso dallo spazio degli stati, sapendo che il valore corrente della variabile aleatoria associata al tempo corrente sia $X_t=i$. Che in formula sarebbe:

$$P\,(X_{t+1}=j|\ X_t=i) = p_{ij}\,(t\ ,\ t+1)$$

Occorre sapere solo lo stato del sistema attuale per calcolare lo stato futuro. Una catena di Markov può essere rappresentata da un *grafico di transizione di stato* o da una *matrice di transizione*. Nell'implementazione presentata in questo testo, verrà utilizzata una matrice di transizione per descrivere il processo stocastico. Fino ad ora abbiamo introdotto il processo di Markov di primo ordine, ma se si utilizzano anche gli eventi passati e non solo l'evento corrente per calcolare la probabilità di transizione si parlerà di Processi di Markov di ordine superiore.

In questo testo ci concentreremo nel calcolo della probabilità di transizione dalla nota corrente ad una possibile nota futura presa dall'insieme dello spazio degli eventi. Le probabilità di transizione saranno calcolate analizzando il Nottingham Music Dataset; la sequenza di note prodotte da questo modello mantiene le stesse probabilità di transizione delle sequenze analizzate nel corpus.
La matrice di transizione è interpretabile come una tabella dove ad ogni colonna e ad ogni riga è associata un elemento dello spazio degli stati. Ogni cella della matrice indica la probabilità di transizione dallo stato della riga allo stato della colonna e, di conseguenza in questo caso, la probabilità di transizione dalla nota della scala cromatica associata alla riga, alla nota associata alla colonna. Quindi per questo lavoro la catena di Markov sarà descritta da una matrice quadrata 12 x 12. Essendo una matrice stocastica, la somma di tutti gli elementi di una riga dovrà essere pari ad uno. Mostriamo una possibile rappresentazione della matrice di transizione con uno spazio degli stati composto solo dalle prime 5 note della scala cromatica espresse in notazione anglosassone.
Si è scelto arbitrariamente di posizionare i valori correnti della nota sulle righe, ed i valori futuri della nota sulle colonne:

Spazio degli stati	Nota A	Nota A#	Nota B	Nota C	Nota C#
Nota A	0.1	0.2	0.5	0.1	0.1
Nota A#	0.05	0.05	0.25	0.25	0.4
Nota B	0.15	0.3	0.15	0.3	0.1
Nota C	0.2	0.1	0.5	0.1	0.1
Nota C#	0.3	0.15	0.15	0.3	0.1

Tabella 1.1: Rappresentazione di una matrice di transizione di una catena di Markov.

In questo esempio l'elemento della matrice $a_{0,1}$= 0.2 in posizione [0,1] sta ad indicare che la probabilità di transizione dalla nota corrente "A" alla nota "A#" nello stato immediatamente successivo è pari a 0.2. Questo dato è stato ottenuto analizzando un corpus di opere.
Se si fosse interessati ad una rappresentazione del grafico di transizione di stato che definisce una catena di Markov, potrete trovare una bella rappresentazione in questo sito web[2]. [9] [10]
[11]

[2] http://setosa.io/ev/markov-chains/

2.3 Implementazione

In questo lavoro si è deciso di emulare il processo creativo tramite l'utilizzo di una catena di Markov di primo ordine progettata in Python. Il lavoro di generazioni della catena di Markov è stato sviluppato servendosi dei famosi moduli di Python: numpy, random e os; oltre alla libreria music21.
Per generare questo processo stocastico abbiamo bisogno di definire uno spazio degli stati detto S, ed uno spazio dei parametri: definiamo come spazio degli stati le dodici note della scala cromatica in notazione anglosassone (A, A#, B, C…). Per definire lo spazio dei parametri in Python ho utilizzato un dizionario chiamato "LetterToNumber" che associ ad ogni nota della scala cromatica (presa come chiave) un numero compreso tra 0 e 11. In questo modo ho indicizzato le dodici note con un numero. Questo passaggio è necessario per poter definire la matrice di transizione utilizzando il modulo *numpy*. Per il processo di composizione, vi è la necessità di affrontare il passaggio inverso: per poter ottenere le note in riferimento agli indici della matrice ho implementato un secondo dizionario chiamato "NumberToLeter" contenente i numeri da 0 a 11 come chiave, riferiti ad ogni nota della scala cromatica. In questo modo ho indicizzato i numeri da 0 a 11 con una nota. Mostriamo la prima parte di codice introdotta fino ad ora:

```
import numpy as np
import random as rm
from music21 import *
import os
import types
import random

LetterToNumber= {'A' : 0, 'A#': 1,'B-':1 ,'B': 2,'C-
':2,'B#':3, 'C':3, 'C#': 4,'D-':4, 'D': 5, 'D#': 6,'E-':6, 'E': 7,'F-
':7,'E#':8, 'F': 8, 'F#': 9,'G-':9, 'G': 10, 'G#':11,'A-':11}
letter=['A','A#','B','C','C#','D','D#','E','F','F#','G','G#']
number= range(12)

NumberToLetter={}

for n in number:
    NumberToLetter[n]=letter[n]
```

Per ottenere la matrice delle probabilità di transizione ho prima creato una matrice intermedia: *la matrice delle frequenze di transizione*. La frequenza di transizione conta, date ad esempio due note "A" e "C#", il numero di volte che all'interno di un brano si è passati dalla nota "A" alla nota "C#". La matrice delle frequenze di transizione, fissata una nota come indice di riga, permette di ottenere

quante volte da quella nota si è passati a tutte le altre 11 note della scala, inserendo ognuno di questi valori come coefficienti nella riga indicizzata dalla nota; si prosegue in questo modo per tutte le 12 note. Per semplicità in seguito parleremo della matrice delle probabilità di transizione come *matrice di probabilità*, e della matrice delle frequenze di transizione come *matrice delle frequenze.* Il calcolo dei coefficienti della matrice di probabilità utilizza i coefficienti delle righe dalla matrice delle frequenze. Per l'esattezza viene applicata la formula della probabilità uniforme (#casi favorevoli / #casi totali). Se si volesse calcolare la probabilità del passaggio da un "A" ad un "D", dovremmo dividere il numero totale di volte in cui è avvenuto questo passaggio con la somma delle volte che si è passati da un "A" a tutte le altre note. Per ottenere la matrice di probabilità, fissata una riga 'i', otteniamo ogni singolo valore di probabilità dividendo il coefficiente della riga i-esima della matrice delle frequenze, con la somma di tutti i valori di quella riga. Inizialmente ho generato le due matrici servendomi del nodulo numpy, entrambe inizializzate a zero come matrici 12 x 12.

```
#definisco lo spazio degli stati della catena di Markov come un dizionario
che associa ad ogni nota un numero da 0 a 11
LetterToNumber= {'A' : 0, 'A#': 1,'B-':1 ,'B': 2,'C-
':2,'B#':3, 'C':3, 'C#': 4,'D-':4, 'D': 5, 'D#': 6,'E-':6, 'E': 7,'F-
':7,'E#':8, 'F': 8, 'F#': 9,'G-':9, 'G': 10, 'G#':11,'A-':11}

letter=['A','A#','B','C','C#','D','D#','E','F','F#','G','G#']

number= range(12)

NumberToLetter={}
for n in number:
    NumberToLetter[n]=letter[n]

#matrice della frequenza di transizione inizializzata come
matrice di zeri
transitionFrequencyMatrix=np.zeros((12,12))
transizionProbabilityMatrix=np.zeros((12,12)) #matrice delle probabilità di
transizione
```

Durante l'analisi di tutto il Nottigham dataset ho incontrato due tipi di notazione utilizzati da music21 per salvare la tipologia della nota: l'utilizzo di "#" per indicare un intervallo di diesis, e di "-" per indicare un intervallo bemolle. Per ovviare a questo problema ho inserito delle chiavi con dei valori duplicati riferite agli intervalli bemolli e diesis.

Per il processo di generazione della matrice delle frequenze ho ottenuto tutto i brani con il metodo utilizzato per la conversione e ho estratto da ognuno di essi la lista delle note con il metodo score.notes. Music21 genera al suo interno una gerarchia di note, il metodo score.flat elimina queste

gerarchie e permette di ottenere tutte le note di uno spartito. Durante l'estrazione delle note ho riscontrato la presenza di due oggetti distinti: note.Note e harmony.ChordSymbol; poiché in questa implementazione siamo interessati solo alle note di ogni spartito ho utilizzato il metodo n.isNote per selezionare solo gli oggetti di tipo note.Note. Con metodo note.name ho ottenuto come stringa il nome della nota che mi ha permesso (tramite l'utilizzo del dizionario creato in precedenza) di ottenere il numero riferito ad essa. Per permettere la comparazione tra due note successive ho creato una variabile temporanea esterna al ciclo che salvasse l'ultima nota estratta; successivamente ho utilizzato la variabile temporanea per incrementare il coefficiente della matrice delle frequenze.

```
abcFiles = os.listdir('./ABC_files')

for abcFile in abcFiles:
  abcFilePath = os.path.abspath(os.path.join('ABC_files', abcFile))

  opus = converter.parseFile(abcFilePath)

  for score in opus.scores:
      temp= None          #creo una variabile temp che salvi l'ultima
                          nota del ciclo per compararla con la successiva
      for n in score.flat.iter.notes:

        if n.isNote: #entra nell'if solo se l'oggetto del ciclo è di tipo
                     note.Note

          noteNumber= LetterToNumber[n.name]

          if temp==None:
            temp=n
          else:
            tempNumber=LetterToNumber[temp.name]

            transitionFrequencyMatrix[tempNumber][noteNumber]=
            (transitionFrequencyMatrix[tempNumber][noteNumber] +1)

            temp=n
```

Occupiamoci ora dell'implementazione della matrice di probabilità. Prima di calcolare ogni singolo coefficiente della matrice si deve ottenere la somma dei valori di ogni riga della matrice delle frequenze; questo è stato fatto utilizzando il metodo matrix.sum(axis=1,type= 'int').

```
rowsArraySum=transitionFrequencyMatrix.sum(axis=1, dtype= 'int')
#ritorna un array contenente la somma degli elementi della matrice raggruppa
ti per riga o per colonna

#axis=0 indica la somma di tutte le colonne, axis=1 indica la somma di tutte
le righe

for row in range(12):
  for column in range(12):
    transitionProbabilityMatrix[row][column]= transitionFrequencyMatrix[row]
[column]/rowsArraySum[row]


rowsArraySumprobability= transitionProbabilityMatrix.sum(axis=1,
dtype= 'int')
print(rowsArraySumprobability) #deve tornare un array contetente solo
numeri uno
```

Il metodo matrix.sum() mi ha permesso inoltre di valutare il corretto funzionamento di questa parte di codice, restituendo un array contenente la somma degli elementi delle righe della matrice di probabilità (essendo una matrice stocastica devono essere tutti uguali ad uno).

2.3.1 Primo modello implementativo

Avendo ora tutti gli strumenti necessari possiamo passare alla parte di implementazione del programma che si occupa della generazione automatica di melodie. Durante questo lavoro ho progettato più approcci successivi per incrementare la qualità del brano composto, esporrò ora la prima implementazione.

L'oggetto Stream è uno dei principali elementi della libreria music21, principale contenitore elementi di notazioni musicale, con numerosi metodi che permettono di creare al suo interno nuovi spartiti. Utilizzeremo questo oggetto per definire un nuovo spartito in cui aggiungeremo delle note servendoci della matrice delle probabilità. Come prima cosa definiamo una nota di partenza assegnandoli una durata di "2/4" cioè come una *semiminima*, come per esempio il Do che nel dizionario fa riferimento al numero 3. Di default l'oggetto stream avrà un metro (lunghezza di ogni battuta) di 4/4. Per impostare un limite superiore al numero di note del nostro spartito ho utilizzato un ciclo *while* servendomi del metodo Stream.notes che fornisce la lista di tutte gli oggetti note.Note contenuti nello spartito. Avendo scelto la nota di partenza, inizializzo una lista come la riga della matrice di probabilità indicizzata da quella nota per ottenere i valori di probabilità di interesse. Scelgo un numero

casuale da zero a uno che mi servirà per scegliere la nota successiva. Ogni riga della matrice è composta da valori di probabilità che sommati danno uno (essendo valori di densità di probabilità); dividiamo l'intervallo tra zero ed uno in intervalli minori, ognuno di ampiezza pari al valore dei singoli coefficienti della riga. Estraiamo un numero casuale tra zero ed uno, la prossima nota scelta sarà quella riferita all' intervallo a cui il numero appartiene. Questo processo è stato implementato tramite l'utilizzo di un ciclo for per scorrere tutti gli intervalli, ed una variabile temporanea utilizzata per il confronto con il numero casuale. Una volta ottenuto l'indice dell'intervallo in cui si trova il numero random, otteniamo la nota corrispondente tramite il dizionario "NumberToLetter". Il metodo Stream.append() permettere di inserire nello spartito la nuova nota e con il parametro interno "type" impostarne la durata (l'impostazione di default è ¼ cioè una semiminima). Per rendere la sequenza di note più gradevole, il primo tentativo fatto è stato quello di creare una lista di possibili durate chiamata noteType1 ed ogni volta che si aggiunge una nota allo spartito impostare la durata di essa come uno dei valori della lista pescato casualmente. Questo introduce qualche cambiamento metrico alla sequenza di note generata, che risulta però parecchio casuale durante l'ascolto.

```
newStream= stream.Stream()  #creo un nuovo spartito
StartingNoteNumber=3
StartingNote= NumberToLetter[StartingNoteNumber]
newStream.append(note.Note(StartingNote,type='half'))
noteType1=['quarter','half','quarter','quarter','quarter','quarter','half','
quarter','quarter','eighth','half']
while len(newStream.notes)<22:
        #scelgo un numero random
  randomNumber= random.random() #restiuisce un intero compreso tra zero
                                e uno
                               #ottengo una riga della matrice
  probabilityMatrixRow=transitionProbabilityMatrix[StartingNoteNumber]


  temp=0 #variabile temporanea pe permettermi la scelta della nota sucessiva

  for index in range(12):

    temp= temp+probabilityMatrixRow[index]
    if randomNumber<= temp:

      nextNote= NumberToLetter[index]
      newStream.append(note.Note(nextNote, type= random.choice(noteType1)))
      #aggiungo la nota trovata allo spartito
      StartingNote=nextNote                #inizializzo la nuova nota di
                                           Partenza come quella corrente
```

```
      StartingNoteNumber=LetterToNumber[StartingNote] #ricavo il suo numero
      break
 newStream.show() # mostro il risultato della composizione
```

2.3.2 Secondo modello implementativo

Nel secondo modello implementativo ho aggiunto due migliorie per rendere il risultato finale maggiormente apprezzabile. La principale miglioria è stata aggiunta creando una sequenza di durate delle note (e.g | 1/4 1/8 1/8 1/4 1/4 | 1/8 1/8 1/8 1/8 1/2 |) ed imponendo allo spartito di seguire ciclicamente quella sequenza; in questo modo possiamo ottenere una struttura metrica ciclica che diminuisce la percezione di casualità nella scelta delle note, rendendo il risultato più gradevole.
Oltre a questa ho deciso di applicare una seconda miglioria allo spartito, ottenuta dalla ricerca di un miglioramento stilistico basandomi su gusti prettamente personali.
Utilizzando il metodo stream.Part() posso dividere il mio spartito in più pentagrammi, che poi saranno riprodotti simultaneamente. Successivamente a questa aggiunta avremo due spartiti creati in modo autonomo (partendo dalla stessa nota) che vengono riprodotti simultaneamente creando un effetto diverso rispetto alla prima implementazione. È necessario notare che il modello markoviano è pensato per la generazione di melodie monofoniche, e dunque la sovrapposizione di due melodie non considera alcuna nozione di armonia. Questa variazione è presentata solo come esempio di ulteriori aggiunte musicali all'output del modello. Ci tengo a precisare che il modello studiato e presentato in questa tesi è stato rappresentato a pieno dalla prima implementazione. Questa scelta arbitraria è stata aggiunta solo come esempio di miglioria del lavoro svolto.
Mostro adesso il codice della seconda implementazione:

```
'''
Secondo metodo implementativo
'''
#creao una lista di durate delle note che si riperà in modo cicliclo
noteTypeDurations=['half','half','quarter','quarter','quarter','quarter','ei
ghth','eighth','eighth','eighth','eighth','eighth','eighth','whole'
]

newStream= stream.Score() # il tempo 4/4 è impostato di default per gli
oggetti Stream


newScore1=stream.Part()
startingNoteNumber=3
startingNote= NumberToLetter[startingNoteNumber]
newScore1.append(note.Note(startingNote,type=noteTypeDurations[0]))
newScore1.id='part1'
for noteIndex in range(1, 60):

    randomNumber= random.random() #restiuisce un intero compreso tra zero
```

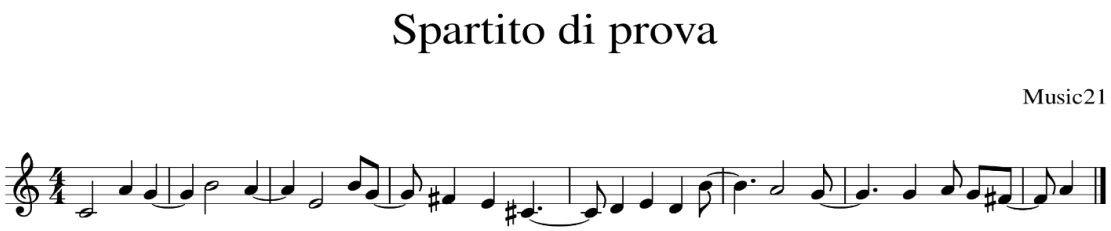
```
    e uno

    probabilityMatrixRow=transitionProbabilityMatrix[startingNoteNumber]


    temp=0 #variabile temporanea pe permettermi la scelta della nota
           successiva
    for index in range(12):

        temp= temp+probabilityMatrixRow[index]
        if randomNumber<= temp:
            duration = noteTypeDurations[noteIndex % len(noteTypeDurations)]

            nextNote= NumberToLetter[index]
            newScore1.append(note.Note(nextNote, type= duration))
            #aggiungo la nota trovata allo spartito
            startingNote=nextNote #inizializzo la nuova nota di partenza
            startingNoteNumber=LetterToNumber[startingNote]
            #ricavo il suo numero
            break
newStream.insert(0,newScore1)


''' creo un secondo spartito partendo sempre dal
‘DO’ che poi unisco al primo '''

newScore2=stream.Part()
startingNoteNumber=3
startingNote= NumberToLetter[startingNoteNumber]
newScore2.append(note.Note(startingNote,type=noteTypeDurations[0]))
newScore2.id='part2'

for noteIndex in range(1, 60):

    randomNumber= random.random() #restiuisce un intero compreso tra zero
    e uno

    probabilityMatrixRow=transitionProbabilityMatrix[startingNoteNumber]

    temp=0 #variabile temporanea pe permettermi la scelta della nota
    successiva
    for index in range(12):

        temp= temp+probabilityMatrixRow[index]
        if randomNumber<= temp:
            duration = noteTypeDurations[noteIndex % len(noteTypeDurations)]
```

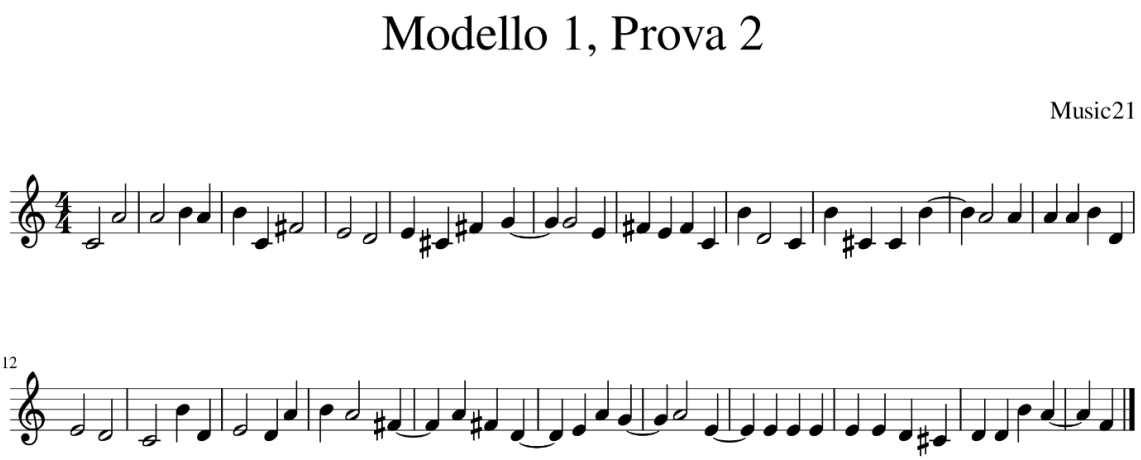
```
            nextNote= NumberToLetter[index]
            newScore2.append(note.Note(nextNote, type= duration))
            #aggiungo la nota trovata allo spartito

            startingNote=nextNote  #inizializzo la nuova nota di partenza
            startingNoteNumber=LetterToNumber[startingNote] #ricavo il suo
            numero
            break


newStream.insert(0,newScore2)
newStream.show()
```

Un rischio di questa implementazione è di poter ottenere note dissonanti nel risultato finale; durante i numerosi tentativi fatti (a parte in qualche caso) ho notato che esse non si presentavano con frequenza così elevato da distorcere totalmente la qualità del brano, poiché entrambi gli spartiti partivano dalla stessa nota impostata come “DO”.

2.4 Risultati finali

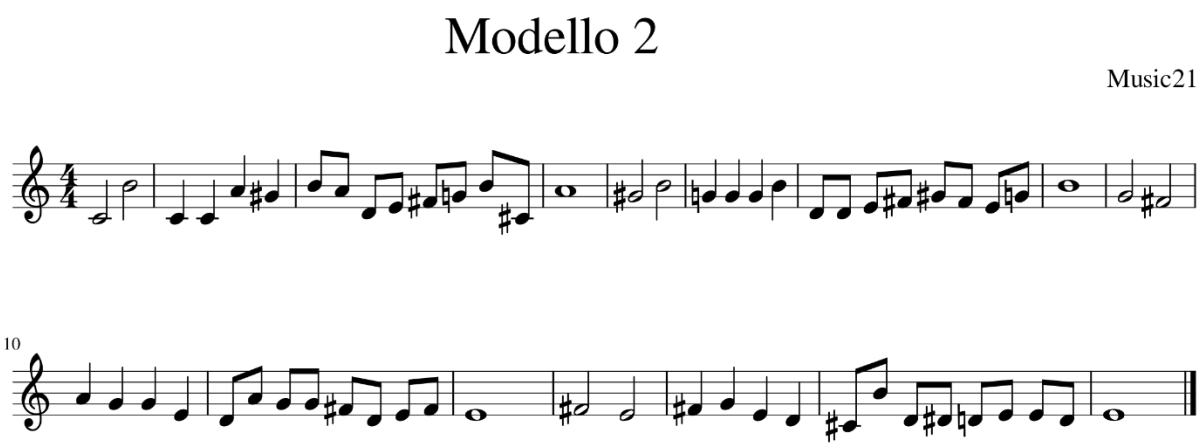
Ecco un esempio di spartito generato con il primo metodo presentato nel paragrafo precedente:

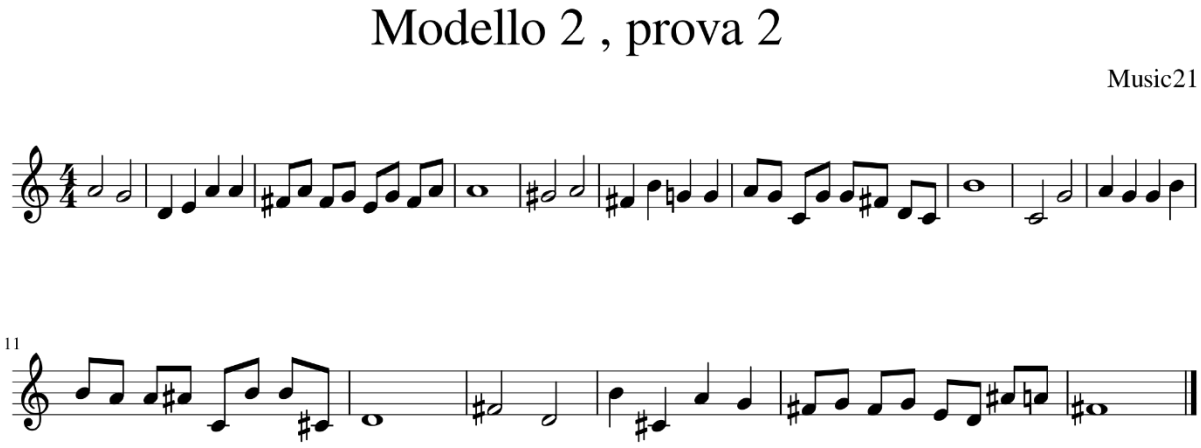




Nel secondo esempio ho tolto dalla lista delle possibili durate delle note le crome (1/8) ed aumentato le note totali da 22 a 64.

In queste pagine vengono mostrati due spartiti generati con la seconda implementazione dove è stato aggiunto un pattern ritmico ciclico per fissare la durata delle note: nel secondo spartito la nota di partenza è stata cambiata da “Do” a “La”

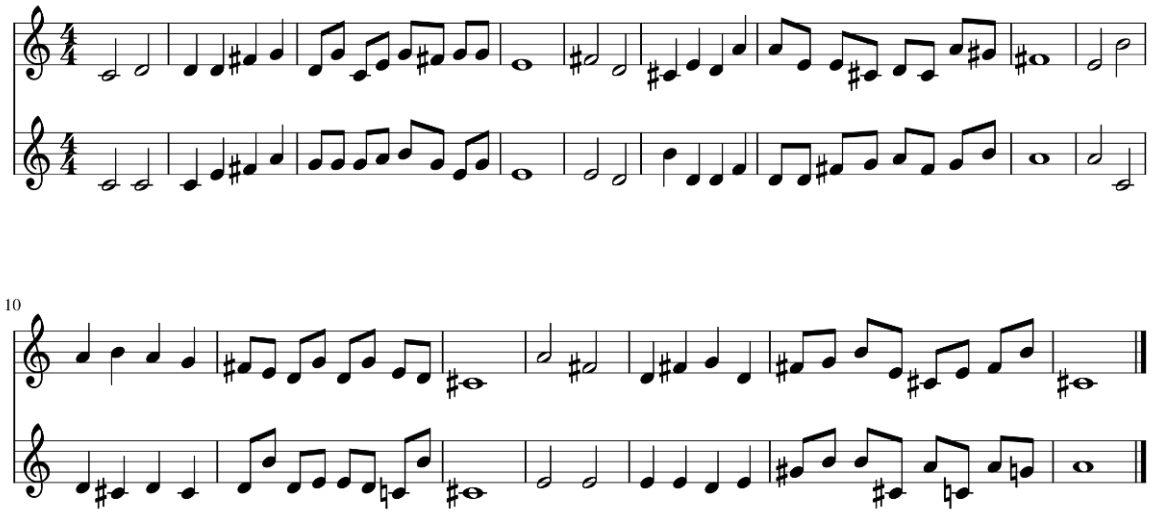




Nella seconda implementazione era stata aggiunta anche la possibilità di unire due battiture separate per ottenere un effetto sonoro differente: in seguito sono presentati due spartiti generati utilizzando questa caratteristica aggiuntiva.

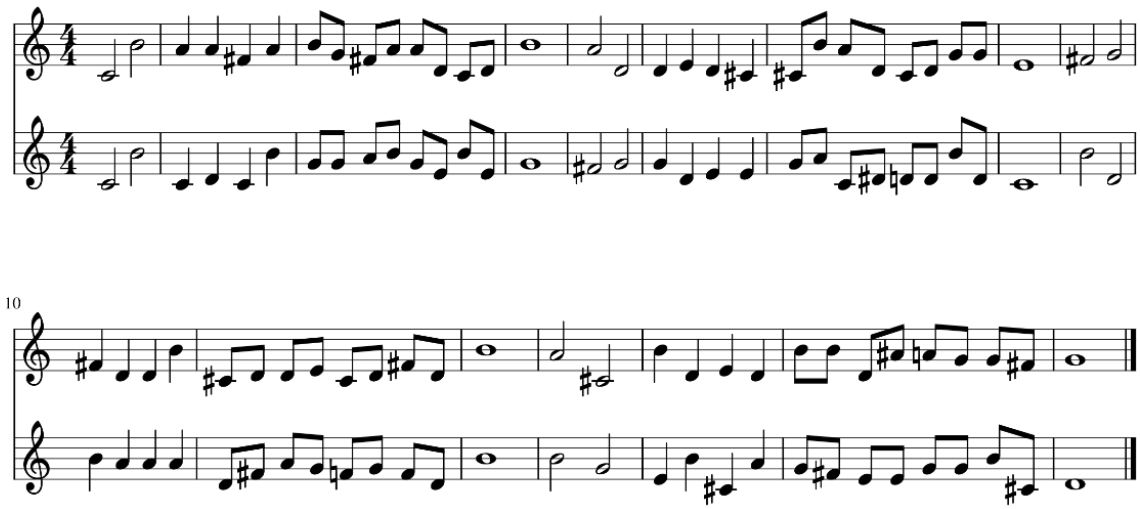
Modello 2, Tentativo 2

Music21



Modello 2, tentativo 1

Music21



3. Conclusioni

In conclusione, posso affermare che il modello di generazione di melodie qui presentato è solo la base di un algoritmo di creatività computazionale. Inserendo altri vincoli di metrica e durata delle note il risultato ottenuto potrebbe migliorare tanto. Un miglioramento possibile per il futuro potrebbe essere introdotto con la generazione automatica di accordi formati dagli intervalli di prima terza e quinta. In questa possibile implementazione si andrebbe a generare uno spartito utilizzando il modello presentato sopra, e in un secondo momento si andrebbe a generare per ogni nota dello spartito l'accordo associato utilizzando il metodo stream.Part() per riprodurre simultaneamente la terza e la quinta nota della nota di partenza. Per la corretta riuscita di questo processo andrebbero aggiunte nozioni e analisi di armonia al lavoro svolto. Un'altra possibile miglioria successiva al processo di Markov potrebbe essere introdotta aggiungendo nozioni di "armonia funzionale" che vadano ad analizzare l'intera sequenza delle note per poter inserire delle armonizzazioni in fase di generazione. In questa possibile implementazione andremmo a generare una semplice sequenza di note bensì una sequenza di accordi basati sulla forma e la tipologia della melodia composta durante il processo di Markov.

Nel campo della creatività computazionale possiamo concludere che i processi stocastici come le catene di Markov hanno grandi potenzialità, soprattutto se accostati a delle nozioni di teoria musicale utilizzate come vincolo in fase di generazione. Accostando al processo di generazione di melodie (tramite le catene di Markov) un algoritmo di intelligenza artificiale basato su un grafo di vincoli teorici (algoritmo della tipologia CSPs, "Constraint satisfaction problems"), si possono ottenere dei risultati sorprendenti e realmente complessi. Per maggiori informazioni riguardo questo modello implementativo rimando a [12].

Bibliografia

[1] M. A. Boden, «Computer Models of Creativity,» *AI Magazine*, vol. 30, n. 3, pp. 24-34, 2009.

[2] B. L. Sturm, «High Noon GMT,» 2 Ottobre 2018. [Online]. Available: https://highnoongmt.wordpress.com/2018/10/02/going-to-use-the-nottingham-music-database/. [Consultato il giorno 06 09 2019].

[3] «MusicXML official web site,» [Online]. Available: https://www.musicxml.com/music-in-musicxml/. [Consultato il giorno 08 Settembre 2019].

[4] S. Mansfield, «The Lesession pages,» [Online]. Available: http://www.lesession.co.uk/abc/abc_notation.htm#intro. [Consultato il giorno Settembre 2019].

[5] Michael Cuthbert, Christopher Ariza, Benjamin Hogue, «Music21 documentation,» [Online]. Available: https://web.mit.edu/music21/doc/.

[6] «MySQL Documentation,» AB, MySQL, 2019-09-07 (revision: 63389). [Online]. Available: https://dev.mysql.com/doc/.

[7] jukedeck, «Git Hub, jikedeck, Nottingham-dataset,» [Online]. Available: https://github.com/jukedeck/nottingham-dataset. [Consultato il giorno 10 Agosto 2019].

[8] Papadopoulos, George, and Geraint Wiggins., «"AI methods for algorithmic composition: A survey, a critical view and future prospects.",» in *AISB Symposium on Musical Creativity, vol. 124*, Edinburgh, UK, 1999, pp. pp. 110-117.

[9] G. Nierhaus, «Chapter 3, Markov models,» in *Algorithmic composition*, 2009, pp. 67-82.

[10] S. Jaiswal, «DataCamp, Markov Chains in Python,» 1 Maggio 2018. [Online]. Available: https://www.datacamp.com/community/tutorials/markov-chains-python-tutorial. [Consultato il giorno settembre 2019].

[11] D. Trevisan, «Dario Trevisan Homepage,» 2016-2017. [Online]. Available: http://people.dm.unipi.it/trevisan/didattica/2016-2017/markov-chains.pdf. [Consultato il giorno 10 Settembre 2019].

[12] MetaCompose: A Compositional Evolutionary, «Metacompose: A compositional evolutionary music composer.,» in *International Conference on Computational Intelligence in Music, Sound, Art and Design*, 2016, March, pp. 202-217.

Appendice

Dopo la fine del lavoro di generazione di melodie ho voluto provare a migliorare il processo di generazione con l'aggiunta di parti accessorie utilizzando le conoscenze apprese su music21. Questa fase, esterna alla fase di ricerca appena presentata, è stata aggiunta solo col tentativo di rielaborare la melodia ottenuta con il processo di Markov per crearne una versione più apprezzabile. Ribadisco che è una parte completamente opzionale che si basa su miglioramenti dettati dal gusto personale.
Col processo fin qui introdotto siamo in grado di generare melodie monofoniche: nella ricerca di ottenere una melodia più complessa ed accattivante è stato implementata una funzione in Python per generare, partendo dalla sequenza di note ottenute con il processo di Markov, la stessa sequenza ad un'ottava maggiore. La sequenza ad un'ottava maggiore verrà aggiunta allo spartito e riprodotta in modo sincrono con la sequenza inziale. Successivamente è stata ampliata la funzione per poter permettere di decidere a quanti intervalli di distanza si volesse replicare la sequenza di partenza: se si volesse replicare a 3 semitoni di distanza si dovrà inserire come input della funzione la nota (in notazione anglosassone) seguita dalla stringa "terza minore". Sono state implementati gli intervalli di terza minore terza maggiore quinta e ottava. Di seguito viene mostrato il codice che definisce la funzione appena introdotta.

```
#definisco lo spazio degli stati della catena di Markov come un dizionario
che associa ad ogni nota un numero da 0 a 11
LetterToNumber= {'A' : 0, 'A#': 1,'B-':1 ,'B': 2,'C-
':2,'B#':3, 'C':3, 'C#': 4,'D-':4, 'D': 5, 'D#': 6,'E-':6, 'E': 7,'F-
':7,'E#':8, 'F': 8, 'F#': 9,'G-':9, 'G': 10, 'G#':11,'A-':11}

letter=['A','A#','B','C','C#','D','D#','E','F','F#','G','G#']

letter5 = {'A': 'A5', 'A#': 'A#5', 'B': 'B5', 'C': 'C5', 'C#': 'C#5', 'D': '
D5', 'D#': 'D#5', 'E': 'E5','F': 'F5','F#': 'F#5', 'G': 'G5','G#': 'G#5' }
# quando si crea una nota senza specificare l'ottava music21 imposta come
numero di ottava di default la quarta.
#per ottenere l'ottava maggiore corrispondente basterà aggiungere alla nota
della sequenza il numero 5 come D5, C#5.


number= range(12)
NumberToLetter={}
for n in number:
    NumberToLetter[n]=letter[n]
```

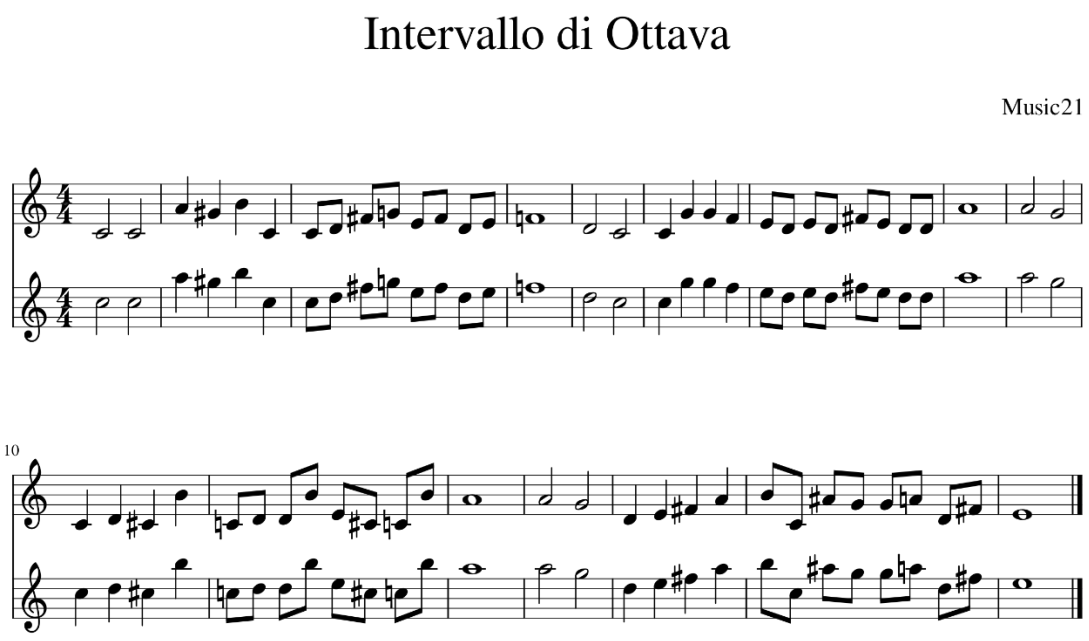
```
#definizione della funzione per ottenere una nota ad un certo intervallo
def calcHarmony(note, harmonyType):
    harmonyTypes = {'terzaMinore': 3, 'terzaMaggiore': 4, 'quinta': 7,
'ottava': 12 }
    noteNumber = LetterToNumber[note]
    harmonyNumber = noteNumber + harmonyTypes[harmonyType]
    if (harmonyType is 'ottava'):
        harmonyNote = letter5[NumberToLetter[harmonyNumber % len(letter5)]]
    else:
        harmonyNote = NumberToLetter[harmonyNumber % len(letter5)]
    return harmonyNote
```

Mostriamo ora le linee di codice riguardanti la generazione:

```
''' Metodo implementativo che sfrutta la funzione calcHarmony per ottenere
una sequenza di note ad un intervallo desiderato'''
newStream= stream.Stream()
newScore1=stream.Part()
newScore2=stream.Part()


noteTypeDurations=['half','half','quarter','quarter','quarter','quarter','ei
ghth','eighth','eighth','eighth','eighth','eighth','eighth','eighth','whole'
]

startingNoteNumber=3
startingNote = NumberToLetter[startingNoteNumber]
n = note.Note(startingNote)
startingNoteHarmony = calcHarmony(startingNote, 'ottava')


newScore1.append(note.Note(startingNote, type=noteTypeDurations[0]))
newScore2.append(note.Note(startingNoteHarmony8, type=noteTypeDurations[0]))


for noteIndex in range(1, 60):
    #scelgo un numero random
    randomNumber = random.random() #restiuisce un intero compreso tra
    zero e uno
    #ottengo una riga della matrice
    probabilityMatrixRow=transitionProbabilityMatrix[startingNoteNumber]
    #print(probabilityMatrixRow)
    #creo un nuovo spartito
    temp=0 #variabile temporanea pe permettermi la scelta della nota
    sucessiva

    for index in range(12):
```

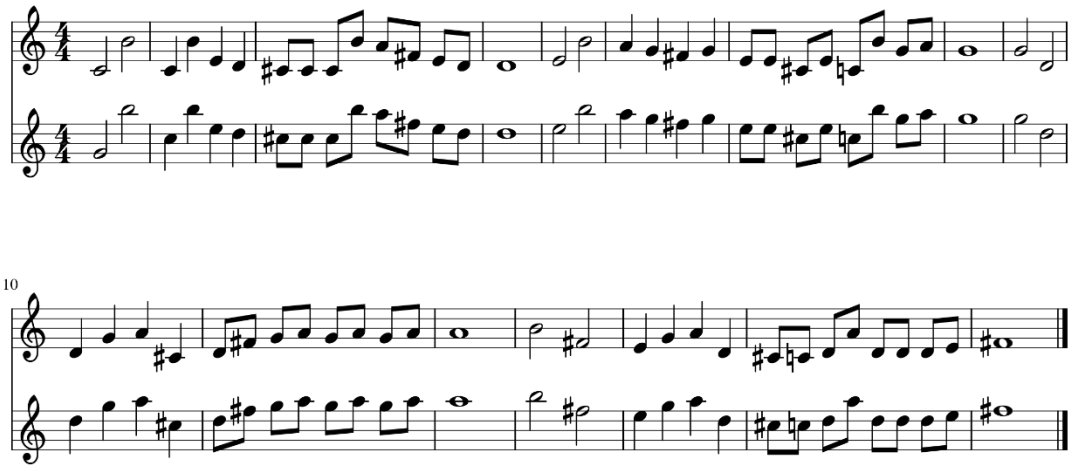
```
        temp= temp+probabilityMatrixRow[index]
        if randomNumber <= temp:
            duration = noteTypeDurations[noteIndex % len(noteTypeDurations)]

            nextNote = NumberToLetter[index]
            newScore1.append(note.Note(nextNote, type=duration))
            #aggiungola nota trovata allo spartito

            harmony = calcHarmony(nextNote, 'ottava')
            newScore2.append(note.Note(harmony, type=duration))


            startingNote=nextNote  # inizializzo la nuova nota di partenza
            startingNoteNumber=LetterToNumber[startingNote]
             #ricavo il suo numero
            break

newStream.insert(0, newScore1)
newStream.insert(0, newScore2)

newStream.show()
```

Riportiamo ora alcuni spartiti ottenuti:



Intervalli di Quinta

Music21



Ringraziamenti

Con la conclusione di questo lavoro si conclude una parte importante della mia vita, in cui ho collezionato sofferenze, delusioni, gioie, stupori, amori ed amicizie.
Non è stato un periodo semplice della mia vita, ma forse il più fruttuoso. Il mio primo ringraziamento va alla facoltà di ingegneria di Padova, che mi ha insegnato quanto sia importate l'impegno costante, la dedizione e la concentrazione per conquistare un obbiettivo che inizialmente ti sembra fuori dalla tua portata. Chi mi conosce sa che vorrei far il musicista da grande quindi è facile chiedersi cosa ci faccio qui a scrivere i ringraziamenti di una laurea in ingegneria. Bhè la vita è strana a volte, ti porta nei posti più disparati senza dare troppe spiegazioni. Ma nonostante tutto sono felice di questi quattro anni, l'ingegneria mi ha reso più forte e stabile mettendo alla prova fino allo stremo le mie energie per permettermi di poter superare ogni ostacolo futuro.
Il secondo grande ringraziamento va ai miei due genitori folli, che non hanno mai smesso di credere in me e di infondermi positività anche nei momenti più bui, senza di loro non sarei mai quello che sono adesso. Grazie davvero Babbo, grazie di cuore Mamma.
Non posso non ringraziare una persona che mi è stata tanto vicino negli ultimi due anni, che si è presa cura di me e della mia casa durante le infinite sessioni d'esame, che mi ha sempre sostenuto ed aiutato a stare bene, grazie a lei sono riuscito a concertarmi al massimo per superare questi ultimi due anni. Quindi grazie infinite ad Alessia, la mia ragazza, per tutto l'affetto dato.
Durante questi 4 anni ho conosciuto molte persone ma sicuramente 3 di queste hanno lasciato il segno più di chiunque altro arricchendo tanto la mia vita universitaria. Tre persone veramente diverse da tutte le altre con cui ho stretto un rapporto fantastico che spero non svanisca mai;
un grazie immenso ai tre miei migliori amici: Samuele, Nick, e Jennifer.
Ringrazio inoltre tutti gli altri miei nuovi amici trovati qui a Padova che mi hanno accompagnato nel mio percorso riempiendolo di momenti pazzi e scatenati: in primis Valeria, Leo, Federico, Vlad, Laura e tutti gli altri della solida compagnia creata in questi anni.

Volevo ringraziare anche una grande persona, sempre disponibile e gentile e presente, che vorrei inserire anche come collaboratore esterno di questo lavoro di tesi, visto l'aiuto informatico che mi ha fornito. Grazie infinite a Gianluca Donato, per tutti i bei momenti vissuti insieme e per l'aiuto concesso.

Infine , ci tengo a ringraziare il relatore di tesi Antonio Rodà, per avermi permesso di fare una tesi in questo ambito a me caro, e soprattutto il co-relatore Filippo Carnovalini per il grande supporto tecnico che mi ha fornito.