INTRODUCTION TO DIGITAL MUSICOLOGY

Fabian C. Moss

2025-01-01

Table of contents

Home

> **Warning**
>
> These pages are work in progress and will be continuously updated.

This page contains material for the course **Introduction to Digital Musicology**, held at Julius-Maximilians-Universität, Würzburg (Germany) in Fall 2025.

The course takes place on Tuesdays from 10 to 12 AM (c.t.) in Room 107 (CIP-Pool), Domerschulstr. 13.

Each week treats a different topic and consists of 45 minutes instruction plus 45 minutes hands-on excercises. Homework and reading material complement the lecture.

> **Note**
>
> If you want to refer to these pages, you can cite them as follows:
> Moss, F. C. (2025). *Introduction to Digital Musicology.* https://fabianmoss.github.io/intro-digimus

Part I.

INTRODUCTION

1. What is Digital Musicology?

> **Goal**
>
> Understanding the meanings of "digital musicology".

1.1. Introduction

Virtual all aspects of our daily lives are increasingly shaped by data and algorithms. The 'digital turn' has also not stopped before academia and science, and most fields are more and more engaging with these new methodologies.

Digital Musicology (DM) is a term that refers to music research that draws on digital data and digital methods to answer its research questions. Stating a concise definition for the field is difficult, and there is no consensus to date. Moreover, a range of similar terms are frequently being used, for example, Computational Musicology.

1.2. Overview of the field

Faced with the fact that no common definition of these terms exist, Kris Shaffer attempted to describe some of the main aspects associated with Computational Musicology (Schaffer 2016). He names the following subfields, which we will briefly discuss in turn (in an adapted order).

1.2.1. Music Encoding

At the beginning of any digital approach to music stands the question how music is to be represented digitally. Music encoding in the narrower sense asks, how *symbolic* musical notations such as Common Western Music Notation (CWMN) can be translated into a computer-readable form. During this course, we will get to know a few fundamental symbolic music encoding formats, namely MIDI, ABC, and MEI. Other important formats, e.g. MusicXML or LilyPond, cannot be covered, unfortunately.

1.2.2. Corpus Studies

Although corpus studies are considered a modern form of music theory, they can look back an an astonishingly long history. Many consider the first corpus study to be Jeppesen's study *The Style of Palestrina and the Dissonance*, where he counts and tabulates occurrences of dissonant vertical intervals in the vocal work of the Renaissance composer (Jeppesen 1927). Further early examples are the dissertations of Budge (1943) and Norman (1945).

In a nutshell, corpus attempt to find regularities in large collections of music data (Shanahan, Burgoyne, and Quinn 2022) in order to draw conclusions about the style of a particular composer, period, or genre, for instance. Corpus studies thus naturally make use of digital data and statistical or algorithmic

methods for their analysis. Corpus studies often try to generalize music theoretical questions from the analysis of individual pieces to entire corpora. They often report their findings in statistical tables or graphical visualizations.

1.2.3. Music Information Retrieval

Music information retrieval (MIR) is the largest part of DM, and research in this domain is directed both at academic as well as industrial contexts. MIR is closer to computer science, and some of its aspects can be understood as CS applied to music. The primary goal is not always a deeper understanding of music, but for instance improvements of recommendation or classification systems (Burgoyne, Fujinaga, and Downie 2015). Recently, the wave of artificial intelligence (AI) as inspired a renewed interest in music generation, and human-computer co-creativity.

1.2.4. Modeling

(mention here also other forms of computational modeling and computational musicology)

The aspects mentioned above are only part of what people nowadays may understand by Computational Musicology. In that sense, Computational Musicology is part of the larger movement of Computational Science, an endeavour that embeds computational methods and thinking into domain-specific areas of research (Wing 2006). If we have this extended view, we also need to include computational models of music perception and cognition, acoustics, and enthnological and anthropological approaches to music as well. However, this would go beyond the scope of this course.

To sum up, what we understand by the term "Digital Musicology" depends on who you ask. What is common to the approaches above, however, is the development or use of digital methods as research tools (word processors don't count, of course). In this course, we focus on the following aspects of Digital Musicology, each of which forms an integral part of its design:

- **Data about music:**
- **Music as data:** How can music, an ephemeral expression of human culture, be captured or represented as digital data on a computer? What kinds of decisions do we have to make? Which musical aspects are retained and which are lost? Is music as data translatable?
- **Working with music data:**
- **Critical Digital Musicology**: What kinds of ethical and societal issues do we touch upon when doing Digital Musicology? Which important questions do we need to consider before planning or executing a research study?
  - e.g. digital vs computational; the latter in 2nd semester
  - digital vs empirical vs quantiative
  - how does DM relate to "traditional" subdivisions of musicology?

Go to the Exercise for Week 1

2. Digital Musicology today

> **Goal**
>
> Acquiring and overview of current activities in Digital Musicology.

TODO: Summarize findings of Inskip and Wiering (2015).

2.1. Istitutions and people

> **Warning**
>
> Obviously, the following institutional lists are incomplete. Luckily for the field of DM, it has grown so extensively that a complete overview is not possible. The selection below is thus not an evaluation, but rather reflects the familiarity of the author with those institutions and the people there.

2.1.1. Germany

- Paderborn/Detmold
- Würzburg
- Erlangen
- Mainz

2.1.2. Europe

- Queen Mary
- Linz
- Barcelona
- DCML
- Amsterdam

2.1.3. USA

- Stanford
- Georgia Tech

2.2. Current research topics

2.3. Central journals and conferences

2.3.1. Journals

- EMR
- Music Perception
- Musicae Scientiae
- Music & Science
- Music Theory Online
- Music Theory Spectrum

2.3.2. Conferences

- IMS Digital Musicology
- ISMIR
- DLfM
- ICMPC
- ICCCM
- CMMR
- CHR
- SMC
- MEC
- DH Conference

2.4. Important tools

- music21
- Verovio

2.5. Study programs and summer schools

**Go to the exercises for Week 2.

3. The history of Digital Musicology

> **Goal**
>
> Knowing the beginnings and the major stages of DM.

Part II.

DATA ABOUT MUSIC

4. RISM metadata

Goal

Learn what metadata are and how to search for music sources on RISM Online.

- What is RISM?
- What is RISM Online?

Exercise

Understand basic SPARQL and design queries via prompting.

5. Spotify and MusicBrainz metadata

> **Goal**
>
> Understand the kind of metadata provided by Spotify vs MusicBrainz.

6. Music and the streaming industry

Goal

Gain first insights into the music market and its workings.

Exercise

Work with sales data.

7. Analyzing song survival

In this session, we will analyze songs from the Billboard 100 charts and trace their 'course of life' in the charts.

The data was obtained from Kaggle, a large community website for data analysis challenges.

As before, we first import the `pandas` library for data analysis and load the data using the `read_csv` fundtion that takes as its main argument the path to the data file, in our case `charts.csv`.

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("data/charts.csv")
```

Inspecting the first 5 lines with the `.head()` method of pandas DataFrames, we obtain an understanding of the structure of the data.

```
df.head()
```

	date	rank	song	artist	last-week	peak-rank	weeks-on-bo
0	2021-11-06	1	Easy On Me	Adele	1.0	1	3
1	2021-11-06	2	Stay	The Kid LAROI & Justin Bieber	2.0	1	16
2	2021-11-06	3	Industry Baby	Lil Nas X & Jack Harlow	3.0	1	14
3	2021-11-06	4	Fancy Like	Walker Hayes	4.0	3	19
4	2021-11-06	5	Bad Habits	Ed Sheeran	5.0	2	18

Think: What do the columns represent? Provide verbal descriptions of their meaning and write it down.

After this general overview, we might want to achieve a slightly deeper understanding. For instance, it is not difficult to interpret the `date` column, but from only the first few entries, we cannot know the temporal extend of our data.

Let's find out what the earliest and latest dates are using the `.min()` and `.max()` methods, respectively.

```
df["date"].min(), df["date"].max()
```

```
('1958-08-04', '2021-11-06')
```

This tells us that the data stored in `charts.csv` runs from August 1958 to November 2021 and thus allows us to trace the movement of songs in the Billboard charts across more than 60 years.

```
# Top artists
df.artist.value_counts()
```

```
artist
Taylor Swift                                                    1023
Elton John                                                       889
Madonna                                                          857
Drake                                                            787
Kenny Chesney                                                    769
                                                                ...
YoungBoy Never Broke Again Featuring Sherhonda Gaulden             1
Drake Featuring Chris Brown                                        1
Kehlani Featuring Jhene Aiko                                       1
DaBaby Featuring A Boogie Wit da Hoodie & London On Da Track       1
The Shins                                                          1
Name: count, Length: 10205, dtype: int64
```

```
# Longest in charts
df.sort_values(by="weeks-on-board", ascending=True).iloc[50_000:]
```

	date	rank	song	artist	last-week	peak-rank	weeks-on-board
213768	1980-11-22	69	Turn And Walk Away	The Babys	79.0	69	2
106440	2001-06-16	41	Fill Me In	Craig David	69.0	41	2
106443	2001-06-16	44	Bootylicious	Destiny's Child	66.0	44	2
106448	2001-06-16	49	All Or Nothing	O-Town	60.0	49	2
213674	1980-11-29	75	My Mother's Eyes	Bette Midler	85.0	75	2
...	...	...	...	...	...	...	...
39148	2014-05-10	49	Radioactive	Imagine Dragons	48.0	3	87
1215	2021-08-14	16	Blinding Lights	The Weeknd	17.0	1	87
1117	2021-08-21	18	Blinding Lights	The Weeknd	16.0	1	88
1020	2021-08-28	21	Blinding Lights	The Weeknd	18.0	1	89
919	2021-09-04	20	Blinding Lights	The Weeknd	21.0	1	90

```
df["date"] = pd.to_datetime(df["date"])
```

```
df[df.artist=="Drake"].song.value_counts()
```

```
song
Hotline Bling     36
God's Plan        36
Controlla         26
Fake Love         25
Nice For What     25
                  ..
Trust Issues       1
Too Much           1
Own It             1
```

```
Tuscan Leather     1
Come Thru          1
Name: count, Length: 108, dtype: int64
```

```
df[df.artist=="Elton John"].song.value_counts()
```

```
song
Candle In The Wind 1997/Something About The Way You Look Tonight    42
Can You Feel The Love Tonight (From "The Lion King")                26
I Guess That's Why They Call It The Blues                           23
The One                                                             22
Candle In The Wind                                                  21
Little Jeannie                                                      21
The Last Song                                                       20
Recover Your Soul                                                   20
Believe                                                             20
Circle Of Life (From "The Lion King")                               20
Blessed                                                             20
Sad Songs (say So Much)                                             19
I Don't Wanna Go On With You Like That                              18
Nikita                                                              18
Bennie And The Jets                                                 18
Mama Can't Buy You Love                                             18
Blue Eyes                                                           18
You Can Make History (Young Again)                                  17
Sacrifice                                                           17
Empty Garden (Hey Hey Johnny)                                       17
Crocodile Rock                                                      17
Goodbye Yellow Brick Road                                           17
I'm Still Standing                                                  16
Club At The End Of The Street                                       16
Simple Life                                                         16
Someday Out Of The Blue                                             15
Don't Let The Sun Go Down On Me                                     15
Island Girl                                                         15
Daniel                                                              15
Rocket Man                                                          15
Healing Hands                                                       15
The Bitch Is Back                                                   14
Who Wears These Shoes?                                              14
Wrap Her Up                                                         14
Sorry Seems To Be The Hardest Word                                  14
Lucy In The Sky With Diamonds                                       14
Your Song                                                           14
Nobody Wins                                                         13
A Word In Spanish                                                   13
Someone Saved My Life Tonight                                       13
You Gotta Love Someone                                              13
In Neon                                                             13
Chloe                                                               13
(Sartorial Eloquence) Don't Ya Wanna Play This Game No More?        12
```

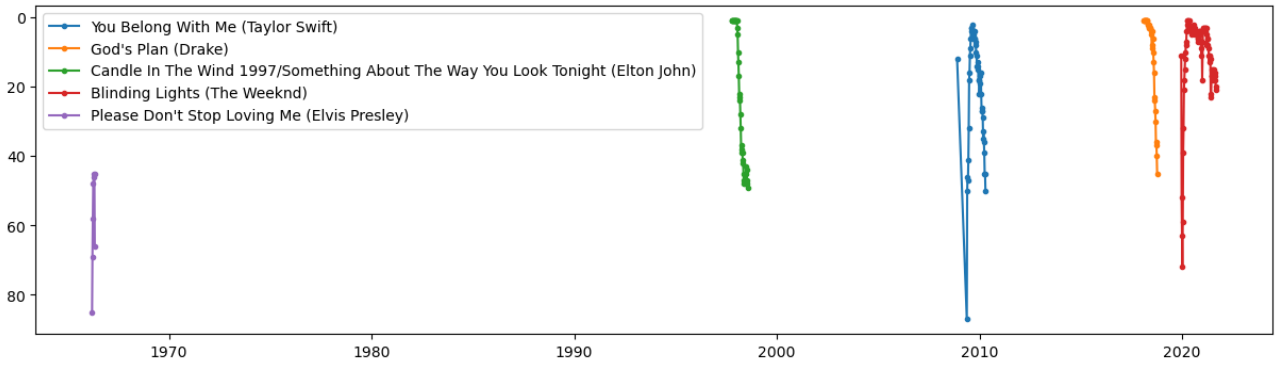
```
Kiss The Bride                                                          12
Saturday Night's Alright For Fighting                                   12
Grow Some Funk Of Your Own/I Feel Like A Bullet (In The Gun Of Robert Ford)    11
Made In England                                                         10
Levon                                                                   10
Victim Of Love                                                          10
Part-Time Love                                                          10
Honky Cat                                                               10
Friends                                                                  9
Heartache All Over The World                                             8
Ego                                                                      8
Tiny Dancer                                                              7
Bite Your Lip (Get up and dance!)                                        6
Border Song                                                              5
Name: count, dtype: int64
```

```python
def chart_performance(artist, song):
    data = df[(df["artist"] == artist) & (df["song"] == song)]
    data = data.sort_values(by="date").reset_index(drop=True)
    data["date_rel"] = pd.to_timedelta(data["date"] - data["date"][0]).dt.days
    return data
```

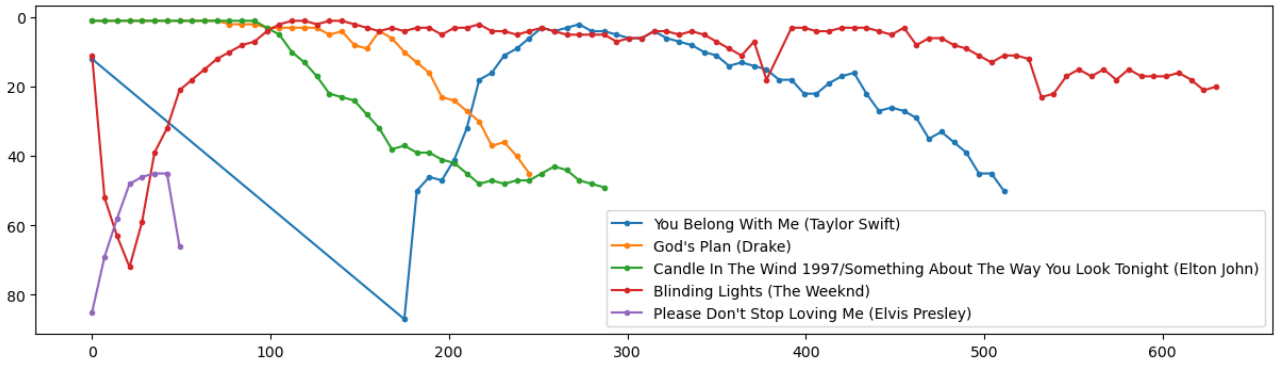
```python
test_cases = {
    "Taylor Swift": "You Belong With Me",
    "Drake": "God's Plan",
    "Elton John": "Candle In The Wind 1997/Something About The Way You Look Tonight",
    "The Weeknd": "Blinding Lights",
    "Elvis Presley": "Please Don't Stop Loving Me"
}
```

```python
taylor = chart_performance("Taylor Swift", "You Belong With Me")
drake = chart_performance("Drake", "God's Plan")
elton = chart_performance("Elton John", "Candle In The Wind 1997/Something About The Way You L
weeknd = chart_performance("The Weeknd", "Blinding Lights")
elvis = chart_performance("Elvis Presley", "Please Don't Stop Loving Me")
```

```python
_, ax = plt.subplots(figsize=(15,4))

for artist, song in test_cases.items():
    data = chart_performance(artist, song)
    x = data["date"].values
    y = data["rank"].values

    ax.plot(x, y, marker=".", label=f"{song} ({artist})")

plt.gca().invert_yaxis()
plt.legend()
plt.show()
```




```
_, ax = plt.subplots(figsize=(15,4))

for artist, song in test_cases.items():
    data = chart_performance(artist, song)
    x = data["date_rel"].values
    y = data["rank"].values

    ax.plot(x, y, marker=".", label=f"{song} ({artist})")

plt.gca().invert_yaxis()
plt.legend()
plt.show()
```




```
# TODO: remove lines for missing weeks (gaps in curves)
# add two cases:
#  - short duration but high peak
#  - long duration but low peak
```

```
# Q: can we predict a song's survival using the features given in the data?
# --> at least introduce notion of training/test data and discuss the epistemological problem
# sources for explanation
```

```
# Try other data: https://www.kaggle.com/datasets/thedevastator/billboard-hot-100-audio-featur
```

```
df_charts = pd.read_csv("Hot Stuff.csv", index_col=0)
df_charts["WeekID"] = pd.to_datetime(df_charts["WeekID"])
```

```
df_charts.head()
```

index	url	WeekID	Week Position	Song
0	http://www.billboard.com/charts/hot-100/1965-0...	1965-07-17	34	Don't Just Stand Th
1	http://www.billboard.com/charts/hot-100/1965-0...	1965-07-24	22	Don't Just Stand Th
2	http://www.billboard.com/charts/hot-100/1965-0...	1965-07-31	14	Don't Just Stand Th
3	http://www.billboard.com/charts/hot-100/1965-0...	1965-08-07	10	Don't Just Stand Th
4	http://www.billboard.com/charts/hot-100/1965-0...	1965-08-14	8	Don't Just Stand Th

```
df_audio = pd.read_csv("Hot 100 Audio Features.csv", index_col=0)
```

```
df_audio.head()
```

index	SongID	Performer	Song
0	-twistin'-White Silver SandsBill Black's Combo	Bill Black's Combo	-twistin'-White Silver Sands
1	¿Dònde Està Santa Claus? (Where Is Santa Claus...	Augie Rios	¿Dònde Està Santa Claus? (
2	.....And Roses And RosesAndy Williams	Andy Williams	.....And Roses And Roses
3	...And Then There Were DrumsSandy Nelson	Sandy Nelson	...And Then There Were Dru
4	...Baby One More TimeBritney Spears	Britney Spears	...Baby One More Time

```
d = df_charts.merge(df_audio)
```

```
d.shape
```

```
(330208, 29)
```

```
d["WeekID"] = pd.to_datetime(d["WeekID"])
```

```
d.sample(10)
```

	url	WeekID	Week Position	Song
275610	http://www.billboard.com/charts/hot-100/1964-0...	1964-06-20	85	My Dreams
189145	http://www.billboard.com/charts/hot-100/2014-0...	2014-02-08	97	Radio
245340	http://www.billboard.com/charts/hot-100/1969-0...	1969-08-02	92	Let's Call It A Day
141579	http://www.billboard.com/charts/hot-100/2009-0...	2009-08-08	3	Knock You Down
117570	http://www.billboard.com/charts/hot-100/1960-0...	1960-02-20	93	Sleepy Lagoon
150750	http://www.billboard.com/charts/hot-100/1966-0...	1966-09-17	36	Flamingo
23582	http://www.billboard.com/charts/hot-100/1985-1...	1985-10-05	63	Soul Kiss
21493	http://www.billboard.com/charts/hot-100/2003-0...	2003-04-12	21	Rock Your Body
71236	http://www.billboard.com/charts/hot-100/2008-1...	2008-12-13	52	My Life
208184	http://www.billboard.com/charts/hot-100/1984-1...	1984-12-15	60	Missing You

```
## BOOTSTRAP!

# d = d.sample(500_000, replace=True)
```

```
d.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 330208 entries, 0 to 330207
Data columns (total 29 columns):
 #   Column                      Non-Null Count   Dtype
---  ------                      --------------   -----
 0   url                         330208 non-null  object
 1   WeekID                      330208 non-null  datetime64[ns]
 2   Week Position               330208 non-null  int64
 3   Song                        330208 non-null  object
 4   Performer                   330208 non-null  object
 5   SongID                      330208 non-null  object
 6   Instance                    330208 non-null  int64
 7   Previous Week Position      298048 non-null  float64
 8   Peak Position               330208 non-null  int64
 9   Weeks on Chart              330208 non-null  int64
 10  spotify_genre               315700 non-null  object
 11  spotify_track_id            287066 non-null  object
 12  spotify_track_preview_url   169915 non-null  object
 13  spotify_track_duration_ms   287066 non-null  float64
 14  spotify_track_explicit      287066 non-null  object
 15  spotify_track_album         287004 non-null  object
 16  danceability                286508 non-null  float64
 17  energy                      286508 non-null  float64
 18  key                         286508 non-null  float64
 19  loudness                    286508 non-null  float64
 20  mode                        286508 non-null  float64
 21  speechiness                 286508 non-null  float64
 22  acousticness                286508 non-null  float64
 23  instrumentalness            286508 non-null  float64
 24  liveness                    286508 non-null  float64
 25  valence                     286508 non-null  float64
 26  tempo                       286508 non-null  float64
 27  time_signature              286508 non-null  float64
 28  spotify_track_popularity    287066 non-null  float64
dtypes: datetime64[ns](1), float64(15), int64(4), object(9)
memory usage: 73.1+ MB
```

```
from IPython.display import Audio, HTML
```

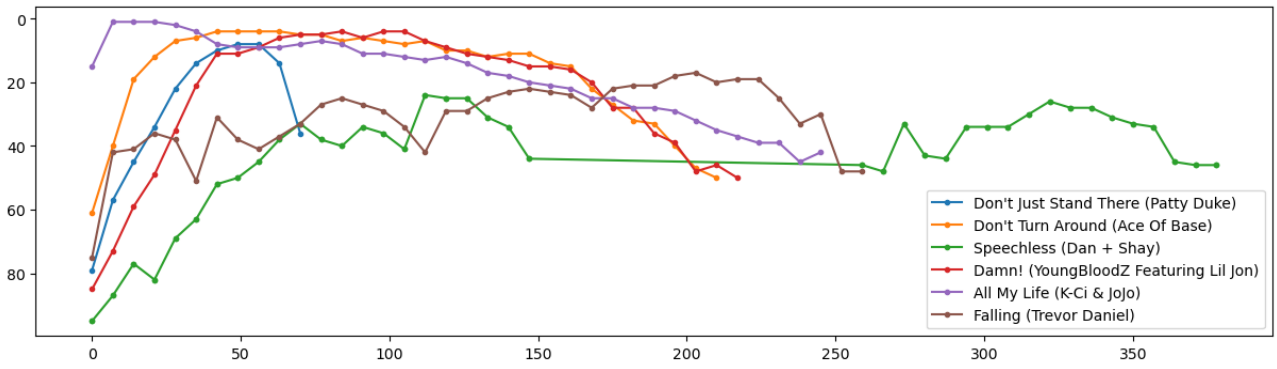
```
Audio(url=d.loc[1000,"spotify_track_preview_url"])
```

```
<IPython.lib.display.Audio object>
```

```
def curves(performer, song):
    data = d[(d.Performer == performer) & (d.Song == song)].sort_values(by="WeekID").reset_ind
    data["date_rel"] = pd.to_timedelta(data["WeekID"] - data["WeekID"][0]).dt.days
    x = data["date_rel"].values # or date_rel or WeekID
    y = data["Week Position"].values
    return x,y
```

```
test_cases2 = {
    "Patty Duke": "Don't Just Stand There",
    "Ace Of Base": "Don't Turn Around",
    "Dan + Shay": "Speechless",
    "YoungBloodZ Featuring Lil Jon": "Damn!",
    "K-Ci & JoJo": "All My Life",
    "Trevor Daniel": "Falling"
}
```

```
_, ax = plt.subplots(figsize=(15,4))

for performer, song in test_cases2.items():
    x,y = curves(performer, song)
    ax.plot(x, y, marker=".", label=f"{song} ({performer})")

plt.gca().invert_yaxis()
plt.legend()
plt.show()
```



Modeling the life of a song in the Top 100:

We assume that once a song has left the Top 100, it is impossible to re-enter (even though that does happen, of course)

1. Each song has a starting rank r_0.
2. For each following week, there is a bernoulli dropout probability θ that determines whether a song remains in the charts.
3.

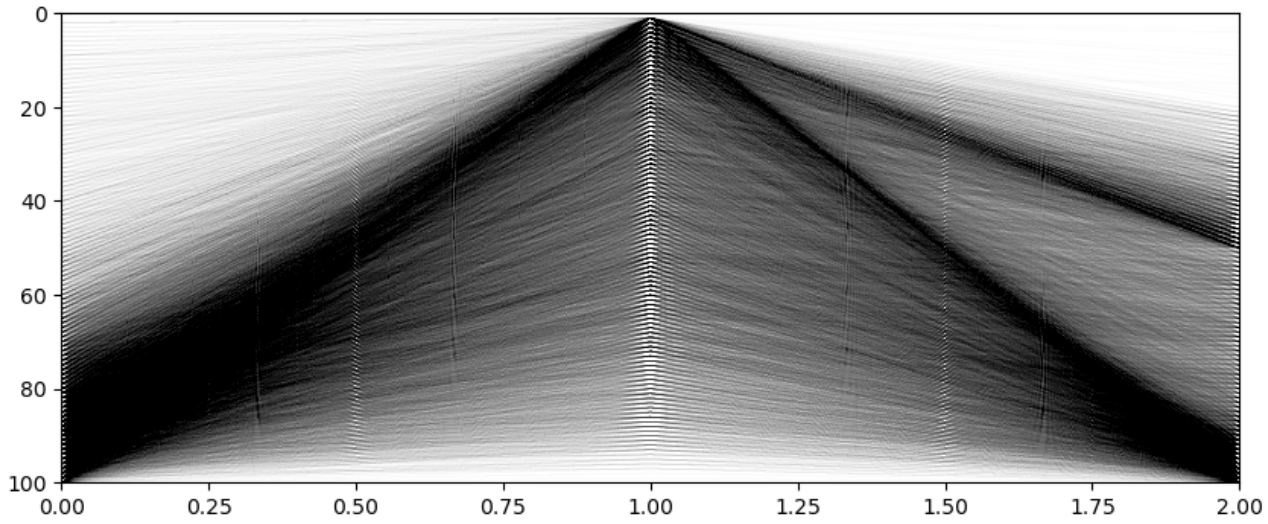
```
# Observation: Genres tend to leave the Top 100 higher than they entered them
```

```
entrances = []
peaks = []
exits = []

for _, group in d.groupby("SongID"):
    weeks = group.sort_values(by="WeekID")["Week Position"].values
    entrances.append(weeks[0])
    peaks.append(weeks.min())
    exits.append(weeks[-1])
```

```
import numpy as np
```

```
# from matplotlib.collections import LineCollection
```

```
_, ax = plt.subplots(figsize=(10,4))

K = len(entrances) + 1

for a, b, c in zip(entrances[:K], peaks[:K], exits[:K]):
    if a != b != c: # remove constants
        ax.plot([0, 1, 2], [a, b, c], c="k", lw=.5, alpha=.01)

plt.xlim(0,2)
plt.ylim(0,100)
plt.gca().invert_yaxis() # smaller is better
plt.savefig("img/rise-decline.png", dpi=600)
plt.show()
```



OBSERVATION: At least 3 types:

- constants
- low in, peak, low out
- low in, peak, mid out

Try to disentangle what causes the difference

Part III.

MUSIC AS DATA

8. Audio

> **Goal**
>
> Understand what an audio signal is and how it is represented digitally .

Clearly, music is related to sound, and artistic commentaries such as John Cage's *4'33* are exceptions that confirm the rule. Thus, in order to understand music and how to encode it, we need a basic understanding of how sound works.

8.1. pure tones

- amplidute A
- frequency f
- limits of hearing, Audible range and volume
- intervals: octave and fifth
- phase ϕ

$$x(t) = A\sin(2\pi ft + \phi)$$

8.2. Harmonics

Adding pure tones and decomposition via Fourier (link to 3b1b)

- Timbre
- Waveform to spectrogram
- reading melodies from a spectrogram

8.3. Digital audio

- Sampling

> **Further reading**
>
> Excellent introductions can be found in Sethares (2005), Müller (2015), and Eerola (2025).

9. MIDI

> **Goal**
>
> Be able to name use cases for MIDI. Translate MIDI numbers to pitches.

We have seen last week that the fundamental frequency of complex tones roughly corresponds to what we perceive as pitch. For many musical styles, pitch is one of the most fundamental components, as it is used to construct melodies and harmonies.

With the invention of electronic instruments in the 20th century, and the possibility of representing continuous sounds as digital signals, it became important for those instruments to interact and communicate. This led to the development of the Musical Instruments Digital Interface (MIDI) format that is still a cornerstone for music information exchange.

MIDI is relatively simple: it consists of so-called messages that have five components:

- Time (in ticks)
- Message (either NOTE ON or NOTE OFF)
- Channel (up to 16 for classical MIDI)
- Note number (integers between 0 and 127)
- Velocity (simulating the speed of pressing down a key, proxy for volume)

In that way, MIDI resembles the much older 'piano roll' representation for musical notes.

ADD PIANO ROLL IMAGE HERE

Exercise:

- Transcribe simple melody to MIDI and plot using `pypianoroll`.
- Download multitrack MIDI from the internet and visualize.
- Describe what can be read from a pianoroll visualization, and what is missing.

10. ABC

> **Goal**
>
> Understanding monophonic symbolic notation.

11. MEI

> **Goal**
>
> Understand basic XML encoding and the skeleton structure of MEI. Understand the relation between CWMN and the MEI music element.

11.1. XML

For a computer, text is merely a string of characters, including 'invisible' characters such as spaces between words and line breaks. But texts are usually more structured than that. For instance, they can contain headings, sections, subsections or paragraphs; they can be printed onto different pages, and they can change their appearance by including **boldfaced** or *italicized* text, for example. If we want to encode these structural aspects of texts in machine-readable formats, they need to be explicitly marked up.

The most popular mark-up language for all sorts of texts is the *eXtensible Markup Language* (XML). XML documents consist of a so-called XML declaration that tells the computer which structuring elements are available and how to interpret them, as well as several tags and their attributes. This is easily explained with an example:

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

<message>
  <greeting>
    Hello world!
  </greeting>
</message>
```

This example contains two types of elements: `message` and `greeting`. Elements work a little bit like parentheses (`(`, `)`) or brackets (`[`, `]`) because there is (almost) always an opening tag and a closing tag. Closing tags contain a slash (`/`).

The example also shows that tags can be hierarchically nested. If the message consisted of several parts, it could be encoded as follows:

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

<message>
  <greeting emotion="joyful">
    Hello world!
  </greeting>
  <politeness>
    Sincerely,
```

```
  </politeness>
  <!-- This is a comment that the computer will ignore -->
</message>
```

We say that `greeting` and `politeness` are *children* of `message` (they are *siblings*), and `greeting` is the parent element.

Elements can contain so-called *attributes* that modify the element. Attributes are listed only in the opening tag of the element. In the example above, the `greeting` element contained the attribute `@emotion` with the *value* `"joyful"`. It is customary to add an `@` when talking about an attribute. Attribute values need to be enclosed in quotation marks (`"`), see also attributes `@version` and `@encoding` of the XML declaration in the first code line.

11.2. Header and body

The basic structure of an MEI document conists of two major parts, the `head` and the `music`, roughly corresponding to metadata describing the source and the music 'itself', respectively.

```
<mei meiversion="5.0">
  <meiHead>
  <!-- metadata goes here -->
  </meiHead>
  <music>
  <!-- description of musical text goes here -->
  </music>
</mei>
```

11.3. A minimal example

11.4. MEI for digital scholarly editions

One of the main reasons for the development of MEI was that existing music encoding standards did not accomodate all needs of music scholars engaged in editing music. Apart from merely representing what is written or printed in a musical score, musicologists understand that musical texts always stand in certain contexts, and that musical sources need commentaries, corrections, and critical treatment more generally. In short, while CWMN is easily represented by other encoding formats such as MusicXML and LilyPond, MEI is the prime candidate for musicological applications due to this critical layer.

11.5. Community

Besides being a digital format for music encoding, MEI is at the same time a community of scholars and practicioners. In fact, the community aspect is very strong, and people very regularly exchange ideas and issues on their Slack communication channel as well in the annual Music Encoding Conference (MEC).

- MuseScore export
- mei friend

Part IV.

WORKING WITH MUSIC DATA

12. Digital music analysis: harmony

> **Goal**
>
> Understand what labeling is and why labels can be useful.

- further MuseScore practice
- segmentation and labeling
- Counting chords, finding cadences

13. Digital music analysis: melody

> **Goal**
>
> Understand how melodic pattern matching works in principle.

- Pattern finding in melodies (Non-Western)

Go to the Exercise for Week X

14. Solos in the *Weimar Jazz Database*

Disclaimer: I am not the expert here!

In this session, we will have a look at is the *Jazzomat Research Project* that contains the *Weimar Jazz Database* (WJazzD). Let us first browse the site.

One of the outcomes of this research project is the freely-available book:

- Pfleiderer, M., Frieler, K., Abeßer, J., Zaddach, W.-G., & Burkhard, B. (Eds.) (2017). Inside the Jazzomat. New Perspectives for Jazz Research. Mainz: Schott Campus (Open Access).

```
import pandas as pd
import numpy as np
import statsmodels.api as sm
import sqlite3

import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("white")
sns.set_context("talk")
```

The WJazzD can be downloaded at https://jazzomat.hfm-weimar.de/download/download.html A local copy of the database is stored at `data/wjazz.db`. We use the `sqlite3` library to connect to this database.

```
conn = sqlite3.connect("data/wjazzd.db")
```

```
conn
```

```
<sqlite3.Connection at 0x201747b7990>
```

We can now use `pandas` to read the data out of the database.

```
solos = pd.read_sql("SELECT * FROM melody", con=conn)
```

The `"SELECT * FROM melody"` means "Select everything from the table 'melody' in the database". Let's look at the first ten entries.

Likewise, we can select the `composition_info` table that contains a lot of metadata for the solos:

```
solos_meta = pd.read_sql("SELECT * from solo_info", con=conn)
```

The `.shape` attribute shows us how many solos are in the database.

```
solos_meta.shape
```

```
(456, 17)
```

The `.sample()` method draws a number of rows at random from a DataFrame.

```
solos_meta.sample(5)
```

	melid	trackid	compid	recordid	performer	title	titleaddon	solopart	instrument	s
429	430	260	236	140	Wayne Shorter	Eighty-One		1	ts	I
440	441	335	295	180	Woody Shaw	Rosewood		1	tp	I
240	241	152	137	76	Johnny Hodges	Bunny		1	as	S
326	327	174	158	87	Miles Davis	Walkin'		1	tp	I
224	225	199	177	103	John Coltrane	Impressions	1963	1	ts	I

The first rows of a DataFrame can be accessed with the `.head()` method…

```
solos.head()
```

	eventid	melid	onset	pitch	duration	period	division	bar	beat	tatum	...	f0_mod	loud_
0	1	1	10.343492	65.0	0.138776	4	1	0	1	1	...		0.126
1	2	1	10.637642	63.0	0.171247	4	4	0	2	1	...		0.349
2	3	1	10.843719	58.0	0.081270	4	4	0	2	4	...		0.094
3	4	1	10.948209	61.0	0.235102	4	1	0	3	1	...		0.521
4	5	1	11.232653	63.0	0.130612	4	1	0	4	1	...		0.560

… and the last rows with the `.tail()` method.

```
solos.tail()
```

	eventid	melid	onset	pitch	duration	period	division	bar	beat	tatum	...	f0_mod
200804	200805	456	63.135057	57.0	0.168345	4	2	53	4	2	...	
200805	200806	456	63.303401	55.0	0.087075	4	3	54	1	1	...	
200806	200807	456	63.390476	57.0	0.191565	4	3	54	1	2	...	slide
200807	200808	456	63.640091	59.0	0.406349	4	1	54	2	1	...	
200808	200809	456	64.058050	52.0	1.433832	4	2	54	3	2	...	vibrato

As we already know, the `.shape` attribute shows the overall size of the table.

```
solos.shape
```

```
(200809, 26)
```

The `solos` table contains 26 columns that cannot be displayed at once. We can have a look at the column names by using the `.columns` attribute.

```
solos.columns
```

```
Index(['eventid', 'melid', 'onset', 'pitch', 'duration', 'period', 'division',
       'bar', 'beat', 'tatum', 'subtatum', 'num', 'denom', 'beatprops',
       'beatdur', 'tatumprops', 'f0_mod', 'loud_max', 'loud_med', 'loud_sd',
       'loud_relpos', 'loud_cent', 'loud_s2b', 'f0_range', 'f0_freq_hz',
       'f0_med_dev'],
      dtype='object')
```

A description of what these columns contain is stated on the website: https://jazzomat.hfm-weimar.de/dbformat/dbformat.html

For our analyses it will be usefull to have also the name of the performer in the `solos` DataFrame. We create a **dictionary** that maps the `melid` (unique identification number for each solo) to the name of the performer.

```
mapper = dict( solos_meta[["melid", "performer"]].values )
mapper
```

```
{1: 'Art Pepper',
 2: 'Art Pepper',
 3: 'Art Pepper',
 4: 'Art Pepper',
 5: 'Art Pepper',
 6: 'Art Pepper',
 7: 'Benny Carter',
 8: 'Benny Carter',
 9: 'Benny Carter',
 10: 'Benny Carter',
 11: 'Benny Carter',
 12: 'Benny Carter',
 13: 'Benny Carter',
 14: 'Benny Goodman',
 15: 'Benny Goodman',
 16: 'Benny Goodman',
 17: 'Benny Goodman',
 18: 'Benny Goodman',
 19: 'Benny Goodman',
 20: 'Benny Goodman',
 21: 'Ben Webster',
 22: 'Ben Webster',
 23: 'Ben Webster',
 24: 'Ben Webster',
 25: 'Ben Webster',
 26: 'Bix Beiderbecke',
 27: 'Bix Beiderbecke',
 28: 'Bix Beiderbecke',
 29: 'Bix Beiderbecke',
 30: 'Bix Beiderbecke',
 31: 'Bob Berg',
```

```
32: 'Bob Berg',
33: 'Bob Berg',
34: 'Bob Berg',
35: 'Bob Berg',
36: 'Bob Berg',
37: 'Bob Berg',
38: 'Branford Marsalis',
39: 'Branford Marsalis',
40: 'Branford Marsalis',
41: 'Branford Marsalis',
42: 'Branford Marsalis',
43: 'Branford Marsalis',
44: 'Buck Clayton',
45: 'Buck Clayton',
46: 'Buck Clayton',
47: 'Cannonball Adderley',
48: 'Cannonball Adderley',
49: 'Cannonball Adderley',
50: 'Cannonball Adderley',
51: 'Cannonball Adderley',
52: 'Charlie Parker',
53: 'Charlie Parker',
54: 'Charlie Parker',
55: 'Charlie Parker',
56: 'Charlie Parker',
57: 'Charlie Parker',
58: 'Charlie Parker',
59: 'Charlie Parker',
60: 'Charlie Parker',
61: 'Charlie Parker',
62: 'Charlie Parker',
63: 'Charlie Parker',
64: 'Charlie Parker',
65: 'Charlie Parker',
66: 'Charlie Parker',
67: 'Charlie Parker',
68: 'Charlie Parker',
69: 'Charlie Shavers',
70: 'Chet Baker',
71: 'Chet Baker',
72: 'Chet Baker',
73: 'Chet Baker',
74: 'Chet Baker',
75: 'Chet Baker',
76: 'Chet Baker',
77: 'Chet Baker',
78: 'Chris Potter',
79: 'Chris Potter',
80: 'Chris Potter',
81: 'Chris Potter',
82: 'Chris Potter',
```

```
83: 'Chris Potter',
84: 'Chris Potter',
85: 'Chu Berry',
86: 'Chu Berry',
87: 'Clifford Brown',
88: 'Clifford Brown',
89: 'Clifford Brown',
90: 'Clifford Brown',
91: 'Clifford Brown',
92: 'Clifford Brown',
93: 'Clifford Brown',
94: 'Clifford Brown',
95: 'Clifford Brown',
96: 'Coleman Hawkins',
97: 'Coleman Hawkins',
98: 'Coleman Hawkins',
99: 'Coleman Hawkins',
100: 'Coleman Hawkins',
101: 'Coleman Hawkins',
102: 'Curtis Fuller',
103: 'Curtis Fuller',
104: 'David Liebman',
105: 'David Liebman',
106: 'David Liebman',
107: 'David Liebman',
108: 'David Liebman',
109: 'David Liebman',
110: 'David Liebman',
111: 'David Liebman',
112: 'David Liebman',
113: 'David Liebman',
114: 'David Liebman',
115: 'David Murray',
116: 'David Murray',
117: 'David Murray',
118: 'David Murray',
119: 'David Murray',
120: 'David Murray',
121: 'Dexter Gordon',
122: 'Dexter Gordon',
123: 'Dexter Gordon',
124: 'Dexter Gordon',
125: 'Dexter Gordon',
126: 'Dexter Gordon',
127: 'Dickie Wells',
128: 'Dickie Wells',
129: 'Dickie Wells',
130: 'Dickie Wells',
131: 'Dickie Wells',
132: 'Dickie Wells',
133: 'Dizzy Gillespie',
```

```
134: 'Dizzy Gillespie',
135: 'Dizzy Gillespie',
136: 'Dizzy Gillespie',
137: 'Dizzy Gillespie',
138: 'Dizzy Gillespie',
139: 'Don Byas',
140: 'Don Byas',
141: 'Don Byas',
142: 'Don Byas',
143: 'Don Byas',
144: 'Don Byas',
145: 'Don Byas',
146: 'Don Byas',
147: 'Don Ellis',
148: 'Don Ellis',
149: 'Don Ellis',
150: 'Don Ellis',
151: 'Don Ellis',
152: 'Don Ellis',
153: 'Eric Dolphy',
154: 'Eric Dolphy',
155: 'Eric Dolphy',
156: 'Eric Dolphy',
157: 'Eric Dolphy',
158: 'Eric Dolphy',
159: 'Fats Navarro',
160: 'Fats Navarro',
161: 'Fats Navarro',
162: 'Fats Navarro',
163: 'Fats Navarro',
164: 'Fats Navarro',
165: 'Freddie Hubbard',
166: 'Freddie Hubbard',
167: 'Freddie Hubbard',
168: 'Freddie Hubbard',
169: 'Freddie Hubbard',
170: 'Freddie Hubbard',
171: 'George Coleman',
172: 'Gerry Mulligan',
173: 'Gerry Mulligan',
174: 'Gerry Mulligan',
175: 'Gerry Mulligan',
176: 'Gerry Mulligan',
177: 'Gerry Mulligan',
178: 'Hank Mobley',
179: 'Hank Mobley',
180: 'Hank Mobley',
181: 'Hank Mobley',
182: 'Harry Edison',
183: 'Henry Allen',
184: 'Herbie Hancock',
```

```
185: 'Herbie Hancock',
186: 'Herbie Hancock',
187: 'Herbie Hancock',
188: 'Herbie Hancock',
189: 'J.C. Higginbotham',
190: 'J.J. Johnson',
191: 'J.J. Johnson',
192: 'J.J. Johnson',
193: 'J.J. Johnson',
194: 'J.J. Johnson',
195: 'J.J. Johnson',
196: 'J.J. Johnson',
197: 'J.J. Johnson',
198: 'Joe Henderson',
199: 'Joe Henderson',
200: 'Joe Henderson',
201: 'Joe Henderson',
202: 'Joe Henderson',
203: 'Joe Henderson',
204: 'Joe Henderson',
205: 'Joe Henderson',
206: 'Joe Lovano',
207: 'Joe Lovano',
208: 'Joe Lovano',
209: 'Joe Lovano',
210: 'Joe Lovano',
211: 'Joe Lovano',
212: 'Joe Lovano',
213: 'Joe Lovano',
214: 'John Abercrombie',
215: 'John Coltrane',
216: 'John Coltrane',
217: 'John Coltrane',
218: 'John Coltrane',
219: 'John Coltrane',
220: 'John Coltrane',
221: 'John Coltrane',
222: 'John Coltrane',
223: 'John Coltrane',
224: 'John Coltrane',
225: 'John Coltrane',
226: 'John Coltrane',
227: 'John Coltrane',
228: 'John Coltrane',
229: 'John Coltrane',
230: 'John Coltrane',
231: 'John Coltrane',
232: 'John Coltrane',
233: 'John Coltrane',
234: 'John Coltrane',
235: 'Johnny Dodds',
```

```
236: 'Johnny Dodds',
237: 'Johnny Dodds',
238: 'Johnny Dodds',
239: 'Johnny Dodds',
240: 'Johnny Dodds',
241: 'Johnny Hodges',
242: 'Johnny Hodges',
243: 'Joshua Redman',
244: 'Joshua Redman',
245: 'Joshua Redman',
246: 'Joshua Redman',
247: 'Joshua Redman',
248: 'Kai Winding',
249: 'Kenny Dorham',
250: 'Kenny Dorham',
251: 'Kenny Dorham',
252: 'Kenny Dorham',
253: 'Kenny Dorham',
254: 'Kenny Dorham',
255: 'Kenny Dorham',
256: 'Kenny Garrett',
257: 'Kenny Garrett',
258: 'Kenny Wheeler',
259: 'Kenny Wheeler',
260: 'Kenny Wheeler',
261: 'Kid Ory',
262: 'Kid Ory',
263: 'Kid Ory',
264: 'Kid Ory',
265: 'Kid Ory',
266: 'Lee Konitz',
267: 'Lee Konitz',
268: 'Lee Konitz',
269: 'Lee Konitz',
270: 'Lee Konitz',
271: 'Lee Konitz',
272: 'Lee Konitz',
273: 'Lee Konitz',
274: 'Lee Morgan',
275: 'Lee Morgan',
276: 'Lee Morgan',
277: 'Lee Morgan',
278: 'Lester Young',
279: 'Lester Young',
280: 'Lester Young',
281: 'Lester Young',
282: 'Lester Young',
283: 'Lester Young',
284: 'Lester Young',
285: 'Lionel Hampton',
286: 'Lionel Hampton',
```

```
287: 'Lionel Hampton',
288: 'Lionel Hampton',
289: 'Lionel Hampton',
290: 'Lionel Hampton',
291: 'Louis Armstrong',
292: 'Louis Armstrong',
293: 'Louis Armstrong',
294: 'Louis Armstrong',
295: 'Louis Armstrong',
296: 'Louis Armstrong',
297: 'Louis Armstrong',
298: 'Louis Armstrong',
299: 'Michael Brecker',
300: 'Michael Brecker',
301: 'Michael Brecker',
302: 'Michael Brecker',
303: 'Michael Brecker',
304: 'Michael Brecker',
305: 'Michael Brecker',
306: 'Michael Brecker',
307: 'Michael Brecker',
308: 'Michael Brecker',
309: 'Miles Davis',
310: 'Miles Davis',
311: 'Miles Davis',
312: 'Miles Davis',
313: 'Miles Davis',
314: 'Miles Davis',
315: 'Miles Davis',
316: 'Miles Davis',
317: 'Miles Davis',
318: 'Miles Davis',
319: 'Miles Davis',
320: 'Miles Davis',
321: 'Miles Davis',
322: 'Miles Davis',
323: 'Miles Davis',
324: 'Miles Davis',
325: 'Miles Davis',
326: 'Miles Davis',
327: 'Miles Davis',
328: 'Milt Jackson',
329: 'Milt Jackson',
330: 'Milt Jackson',
331: 'Milt Jackson',
332: 'Milt Jackson',
333: 'Milt Jackson',
334: 'Nat Adderley',
335: 'Nat Adderley',
336: 'Ornette Coleman',
337: 'Ornette Coleman',
```

```
338: 'Ornette Coleman',
339: 'Ornette Coleman',
340: 'Ornette Coleman',
341: 'Pat Martino',
342: 'Pat Metheny',
343: 'Pat Metheny',
344: 'Pat Metheny',
345: 'Pat Metheny',
346: 'Paul Desmond',
347: 'Paul Desmond',
348: 'Paul Desmond',
349: 'Paul Desmond',
350: 'Paul Desmond',
351: 'Paul Desmond',
352: 'Paul Desmond',
353: 'Paul Desmond',
354: 'Pepper Adams',
355: 'Pepper Adams',
356: 'Pepper Adams',
357: 'Pepper Adams',
358: 'Pepper Adams',
359: 'Phil Woods',
360: 'Phil Woods',
361: 'Phil Woods',
362: 'Phil Woods',
363: 'Phil Woods',
364: 'Phil Woods',
365: 'Red Garland',
366: 'Rex Stewart',
367: 'Roy Eldridge',
368: 'Roy Eldridge',
369: 'Roy Eldridge',
370: 'Roy Eldridge',
371: 'Roy Eldridge',
372: 'Roy Eldridge',
373: 'Sidney Bechet',
374: 'Sidney Bechet',
375: 'Sidney Bechet',
376: 'Sidney Bechet',
377: 'Sidney Bechet',
378: 'Sonny Rollins',
379: 'Sonny Rollins',
380: 'Sonny Rollins',
381: 'Sonny Rollins',
382: 'Sonny Rollins',
383: 'Sonny Rollins',
384: 'Sonny Rollins',
385: 'Sonny Rollins',
386: 'Sonny Rollins',
387: 'Sonny Rollins',
388: 'Sonny Rollins',
```

```
389: 'Sonny Rollins',
390: 'Sonny Rollins',
391: 'Sonny Stitt',
392: 'Sonny Stitt',
393: 'Sonny Stitt',
394: 'Sonny Stitt',
395: 'Sonny Stitt',
396: 'Sonny Stitt',
397: 'Stan Getz',
398: 'Stan Getz',
399: 'Stan Getz',
400: 'Stan Getz',
401: 'Stan Getz',
402: 'Stan Getz',
403: 'Steve Coleman',
404: 'Steve Coleman',
405: 'Steve Coleman',
406: 'Steve Coleman',
407: 'Steve Coleman',
408: 'Steve Coleman',
409: 'Steve Coleman',
410: 'Steve Coleman',
411: 'Steve Coleman',
412: 'Steve Coleman',
413: 'Steve Lacy',
414: 'Steve Lacy',
415: 'Steve Lacy',
416: 'Steve Lacy',
417: 'Steve Lacy',
418: 'Steve Lacy',
419: 'Steve Turre',
420: 'Steve Turre',
421: 'Steve Turre',
422: 'Von Freeman',
423: 'Warne Marsh',
424: 'Warne Marsh',
425: 'Warne Marsh',
426: 'Wayne Shorter',
427: 'Wayne Shorter',
428: 'Wayne Shorter',
429: 'Wayne Shorter',
430: 'Wayne Shorter',
431: 'Wayne Shorter',
432: 'Wayne Shorter',
433: 'Wayne Shorter',
434: 'Wayne Shorter',
435: 'Wayne Shorter',
436: 'Woody Shaw',
437: 'Woody Shaw',
438: 'Woody Shaw',
439: 'Woody Shaw',
```

```
 440: 'Woody Shaw',
 441: 'Woody Shaw',
 442: 'Woody Shaw',
 443: 'Woody Shaw',
 444: 'Wynton Marsalis',
 445: 'Wynton Marsalis',
 446: 'Wynton Marsalis',
 447: 'Wynton Marsalis',
 448: 'Wynton Marsalis',
 449: 'Wynton Marsalis',
 450: 'Wynton Marsalis',
 451: 'Zoot Sims',
 452: 'Zoot Sims',
 453: 'Zoot Sims',
 454: 'Zoot Sims',
 455: 'Zoot Sims',
 456: 'Zoot Sims'}
```

We can now use this dictionary to create a new column `performer` in the `solos` DataFrame.

```
solos["performer"] = solos["melid"].map(mapper)
```

```
solos.head()
```

	eventid	melid	onset	pitch	duration	period	division	bar	beat	tatum	...	loud_max	lou
0	1	1	10.343492	65.0	0.138776	4	1	0	1	1	...	0.126209	66.
1	2	1	10.637642	63.0	0.171247	4	4	0	2	1	...	0.349751	69.
2	3	1	10.843719	58.0	0.081270	4	4	0	2	4	...	0.094051	66.
3	4	1	10.948209	61.0	0.235102	4	1	0	3	1	...	0.521187	66.
4	5	1	11.232653	63.0	0.130612	4	1	0	4	1	...	0.560737	71.

```
solos.tail()
```

	eventid	melid	onset	pitch	duration	period	division	bar	beat	tatum	...	loud_max
200804	200805	456	63.135057	57.0	0.168345	4	2	53	4	2	...	1.113380
200805	200806	456	63.303401	55.0	0.087075	4	3	54	1	1	...	0.491496
200806	200807	456	63.390476	57.0	0.191565	4	3	54	1	2	...	1.187058
200807	200808	456	63.640091	59.0	0.406349	4	1	54	2	1	...	0.972676
200808	200809	456	64.058050	52.0	1.433832	4	2	54	3	2	...	0.368321

14.1. Melodic arc?

Does the melodic arc also appear in the Jazz solos?

```
def notelist(melid):

    solo = solos[solos["melid"] == melid]

    solo = solo[["pitch", "duration"]]
    solo["onset"] = solo["duration"].cumsum()
    return solo
```

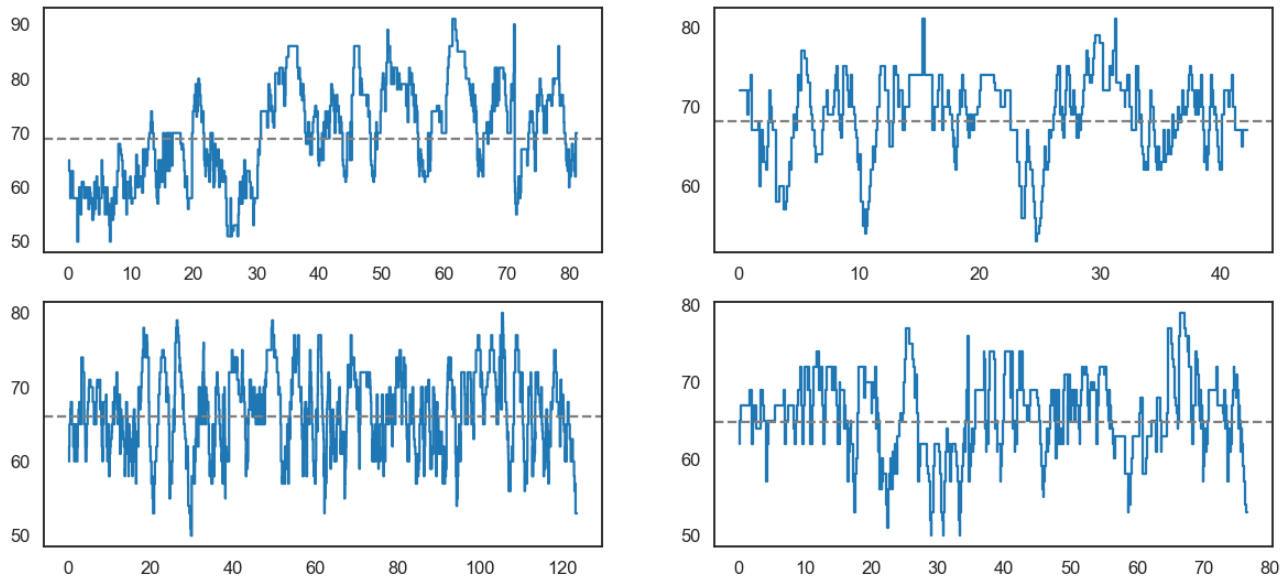
```
notelist(1)
```

	pitch	duration	onset
0	65.0	0.138776	0.138776
1	63.0	0.171247	0.310023
2	58.0	0.081270	0.391293
3	61.0	0.235102	0.626395
4	63.0	0.130612	0.757007
...	...	...	...
525	66.0	0.137143	80.645238
526	65.0	0.101587	80.746825
527	63.0	0.104490	80.851315
528	62.0	0.110295	80.961610
529	70.0	0.187211	81.148821

```
def plot_melodic_profile(notelist, ax=None, c=None, mean=False, Z=False, sections=False, stand

    if ax == None:
        ax = plt.gca()

    if standardized:
        x = notelist["Rel. Onset"]
        y = notelist["Rel. MIDI Pitch"]
    else:
        x = notelist["onset"]
        y = notelist["pitch"]

    ax.step(x,y, color=c)

    if mean:
        ax.axhline(y.mean(), color="gray", linestyle="--")

    if sections:
        for l in [ x.max() * i for i in [ 1/4, 1/2, 3/4] ]:
            ax.axvline(l, color="gray", linewidth=1, linestyle="--")
```

```
fig, axes = plt.subplots(2,2, figsize=(20,9))
axes = axes.flatten()

plot_melodic_profile(notelist(1), ax=axes[0], mean=True)
plot_melodic_profile(notelist(77), ax=axes[1], mean=True)
```

```
plot_melodic_profile(notelist(50), ax=axes[2], mean=True)
plot_melodic_profile(notelist(233), ax=axes[3], mean=True)
```



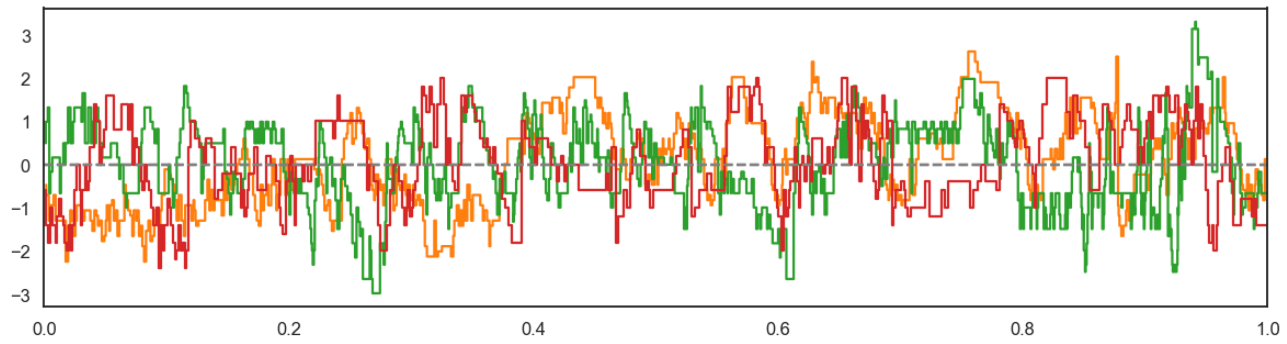

```
def standardize(notelist):
    """
    Takes a notelist as input and returns a standardized version.
    """

    notelist["Rel. MIDI Pitch"] = (notelist["pitch"] - notelist["pitch"].mean()) / notelist["p
    notelist["Rel. Duration"] = notelist["duration"] / notelist["duration"].sum()
    notelist["Rel. Onset"] = notelist["onset"] / notelist["onset"].max()

    return notelist
```

```
standardize(notelist(1))
```

	pitch	duration	onset	Rel. MIDI Pitch	Rel. Duration	Rel. Onset
0	65.0	0.138776	0.138776	-0.460594	0.001710	0.001710
1	63.0	0.171247	0.310023	-0.697714	0.002110	0.003820
2	58.0	0.081270	0.391293	-1.290513	0.001001	0.004822
3	61.0	0.235102	0.626395	-0.934833	0.002897	0.007719
4	63.0	0.130612	0.757007	-0.697714	0.001610	0.009329
...	...	...	...	...	...	...
525	66.0	0.137143	80.645238	-0.342034	0.001690	0.993794
526	65.0	0.101587	80.746825	-0.460594	0.001252	0.995046
527	63.0	0.104490	80.851315	-0.697714	0.001288	0.996334
528	62.0	0.110295	80.961610	-0.816274	0.001359	0.997693
529	70.0	0.187211	81.148821	0.132205	0.002307	1.000000

```
fig, ax = plt.subplots(figsize=(20,5))
```

```
for i in range(4):
    plot_melodic_profile(standardize(notelist(i)),
                         mean=True,
                         standardized=True)
plt.xlim(0,1)
plt.show()
```




```
big_df = pd.concat([standardize(notelist(i)) for i in range(solos_meta.shape[0])])
```

```
solos
```

	eventid	melid	onset	pitch	duration	period	division	bar	beat	tatum	...	loud_max
0	1	1	10.343492	65.0	0.138776	4	1	0	1	1	...	0.126209
1	2	1	10.637642	63.0	0.171247	4	4	0	2	1	...	0.349751
2	3	1	10.843719	58.0	0.081270	4	4	0	2	4	...	0.094051
3	4	1	10.948209	61.0	0.235102	4	1	0	3	1	...	0.521187
4	5	1	11.232653	63.0	0.130612	4	1	0	4	1	...	0.560737
...	...	...	...	...	...	...	...	...	...	...	...	...
200804	200805	456	63.135057	57.0	0.168345	4	2	53	4	2	...	1.113380
200805	200806	456	63.303401	55.0	0.087075	4	3	54	1	1	...	0.491496
200806	200807	456	63.390476	57.0	0.191565	4	3	54	1	2	...	1.187058
200807	200808	456	63.640091	59.0	0.406349	4	1	54	2	1	...	0.972676
200808	200809	456	64.058050	52.0	1.433832	4	2	54	3	2	...	0.368321

```
big_df
```

	pitch	duration	onset	Rel. MIDI Pitch	Rel. Duration	Rel. Onset
0	65.0	0.138776	0.138776	-0.460594	0.001710	0.001710
1	63.0	0.171247	0.310023	-0.697714	0.002110	0.003820
2	58.0	0.081270	0.391293	-1.290513	0.001001	0.004822
3	61.0	0.235102	0.626395	-0.934833	0.002897	0.007719
4	63.0	0.130612	0.757007	-0.697714	0.001610	0.009329
...	...	...	...	...	...	...
200585	62.0	0.870748	68.588934	0.014206	0.012540	0.987794
200586	57.0	0.133515	68.722449	-0.896471	0.001923	0.989717
200587	62.0	0.139320	68.861769	0.014206	0.002006	0.991723

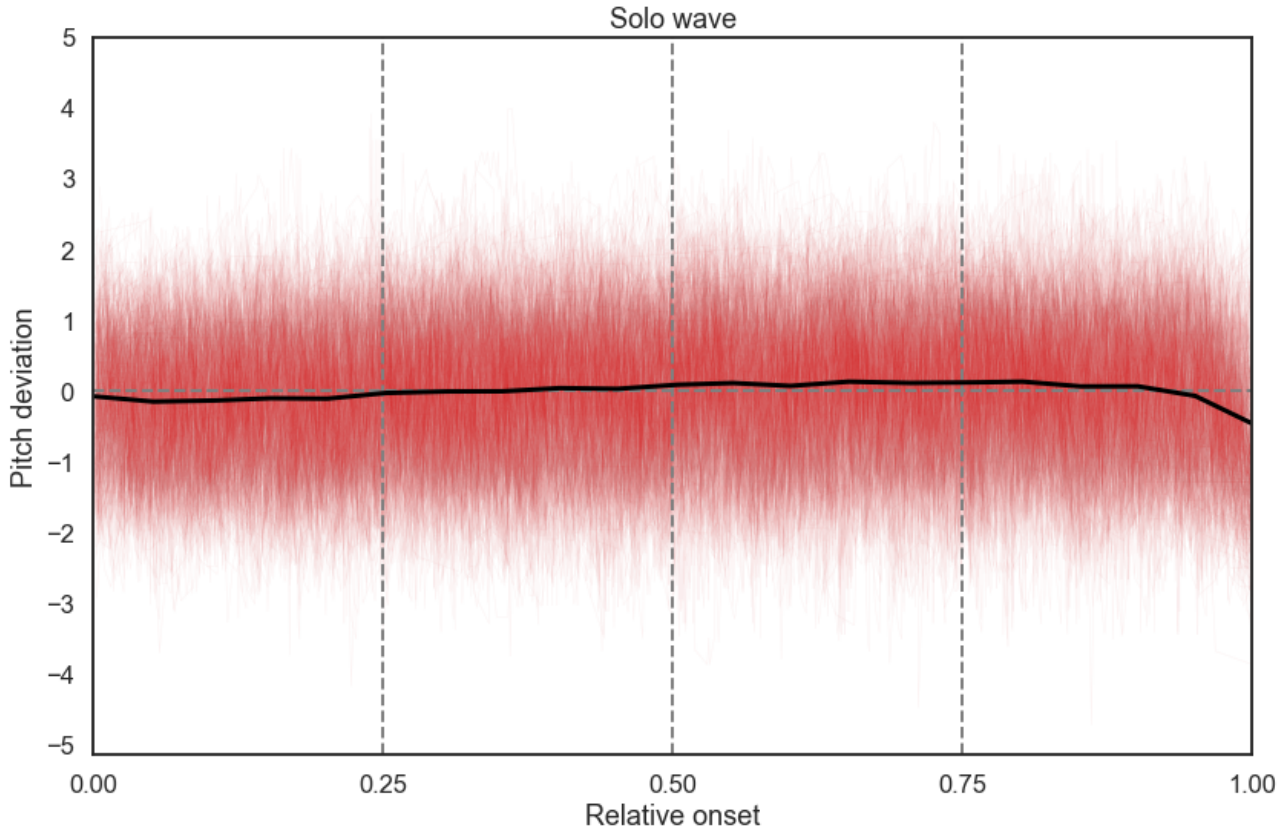
	pitch	duration	onset	Rel. MIDI Pitch	Rel. Duration	Rel. Onset
200588	61.0	0.133515	68.995283	-0.167930	0.001923	0.993646
200589	60.0	0.441179	69.436463	-0.350065	0.006354	1.000000

```
solos_meta["performer"].unique()
```

```
array(['Art Pepper', 'Benny Carter', 'Benny Goodman', 'Ben Webster',
       'Bix Beiderbecke', 'Bob Berg', 'Branford Marsalis', 'Buck Clayton',
       'Cannonball Adderley', 'Charlie Parker', 'Charlie Shavers',
       'Chet Baker', 'Chris Potter', 'Chu Berry', 'Clifford Brown',
       'Coleman Hawkins', 'Curtis Fuller', 'David Liebman',
       'David Murray', 'Dexter Gordon', 'Dickie Wells', 'Dizzy Gillespie',
       'Don Byas', 'Don Ellis', 'Eric Dolphy', 'Fats Navarro',
       'Freddie Hubbard', 'George Coleman', 'Gerry Mulligan',
       'Hank Mobley', 'Harry Edison', 'Henry Allen', 'Herbie Hancock',
       'J.C. Higginbotham', 'J.J. Johnson', 'Joe Henderson', 'Joe Lovano',
       'John Abercrombie', 'John Coltrane', 'Johnny Dodds',
       'Johnny Hodges', 'Joshua Redman', 'Kai Winding', 'Kenny Dorham',
       'Kenny Garrett', 'Kenny Wheeler', 'Kid Ory', 'Lee Konitz',
       'Lee Morgan', 'Lester Young', 'Lionel Hampton', 'Louis Armstrong',
       'Michael Brecker', 'Miles Davis', 'Milt Jackson', 'Nat Adderley',
       'Ornette Coleman', 'Pat Martino', 'Pat Metheny', 'Paul Desmond',
       'Pepper Adams', 'Phil Woods', 'Red Garland', 'Rex Stewart',
       'Roy Eldridge', 'Sidney Bechet', 'Sonny Rollins', 'Sonny Stitt',
       'Stan Getz', 'Steve Coleman', 'Steve Lacy', 'Steve Turre',
       'Von Freeman', 'Warne Marsh', 'Wayne Shorter', 'Woody Shaw',
       'Wynton Marsalis', 'Zoot Sims'], dtype=object)
```

```
%%time

fig, ax = plt.subplots(figsize=(12,8))

artists = ["Louis Armstrong"]

# for i, (artist, group) in enumerate(solos.groupby("performer")):
#     if artist in artists:
#         for j, group in group.groupby("melid"):
#             solo = standardize(notelist(j))
#             x = solo["Rel. Onset"]
#             y = solo["Rel. MIDI Pitch"]
#             ax. plot(x,y, lw=.5, c="tab:red", alpha=.5)

for ID in range(solos_meta.shape[0]):
    solo = standardize(notelist(ID))
    x = solo["Rel. Onset"]
    y = solo["Rel. MIDI Pitch"]
    ax. plot(x,y, lw=.5, c="tab:red", alpha=.05)

ax.axvline(.25, lw=2, ls="--", c="gray")
```

```
ax.axvline(.5, lw=2, ls="--", c="gray")
ax.axvline(.75, lw=2, ls="--", c="gray")
ax.axhline(0, lw=2, ls="--", c="gray")

lowess = sm.nonparametric.lowess
big_x = big_df["Rel. Onset"]
big_y = big_df["Rel. MIDI Pitch"]
big_z = lowess(big_y, big_x, frac=1/10, delta=1/20)
ax.plot(big_z[:,0], big_z[:,1], c="black", lw=3)

plt.title("Solo wave")
plt.xlabel("Relative onset")
plt.ylabel("Pitch deviation")
plt.xticks(np.linspace(0,1,5))
plt.yticks(np.linspace(-5,5,11))
plt.xlim(0,1)

plt.tight_layout()
plt.savefig("img/jazz_melodic_arc.png")
plt.show()
```




```
Wall time: 7.76 s
```

14.2. Pitch vs loudness

Above we have already analyzed some melodic profiles and seen that, on average, the Jazz solos tend not to follow the melodic arch on a global scale. Now, we ask whether the pitch of the notes in the solos are related to another important feature of performance: loudness. The WJazzD contains several measures for loudness (compare the columns in the `solos` DataFrame). Here, we focus on the "Median loudness" which is stored in the `loud_med` column.

Let us look at an example.

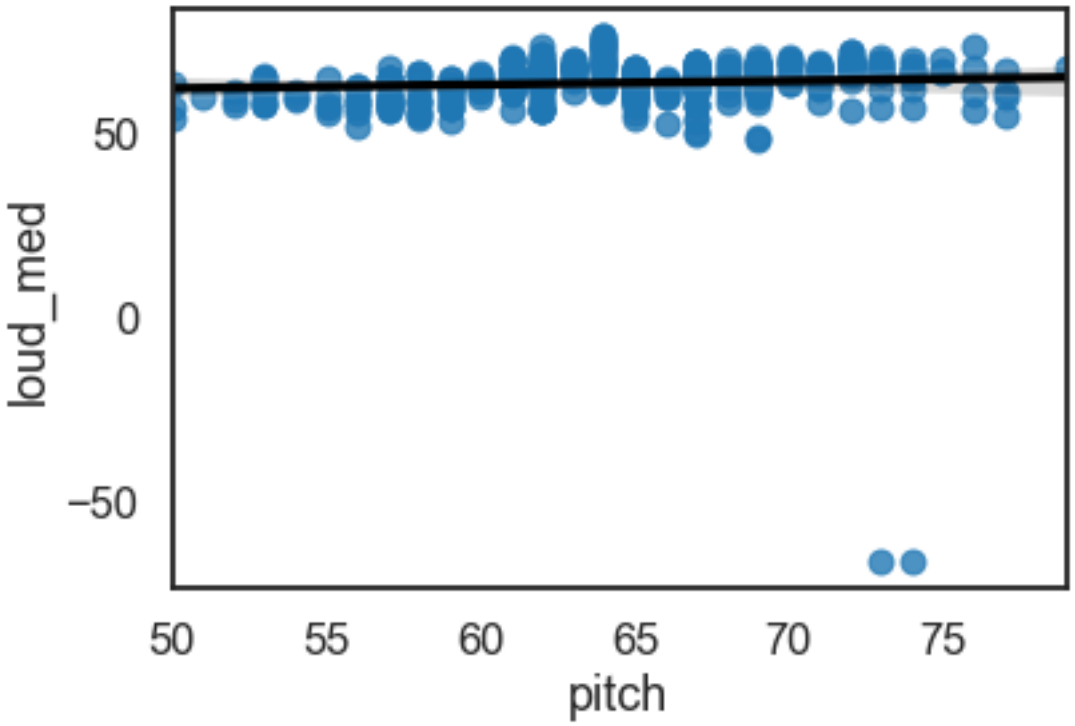
```
example_solo = solos[ solos["melid"] == 233 ][["pitch", "loud_med"]]
```

```
example_solo
```

	pitch	loud_med
115075	62.0	67.082700
115076	65.0	65.345677
115077	67.0	66.323539
115078	69.0	69.204257
115079	62.0	69.059581
...	...	...
115549	59.0	61.083907
115550	57.0	58.345887
115551	55.0	65.132786
115552	54.0	59.595735
115553	53.0	58.390132

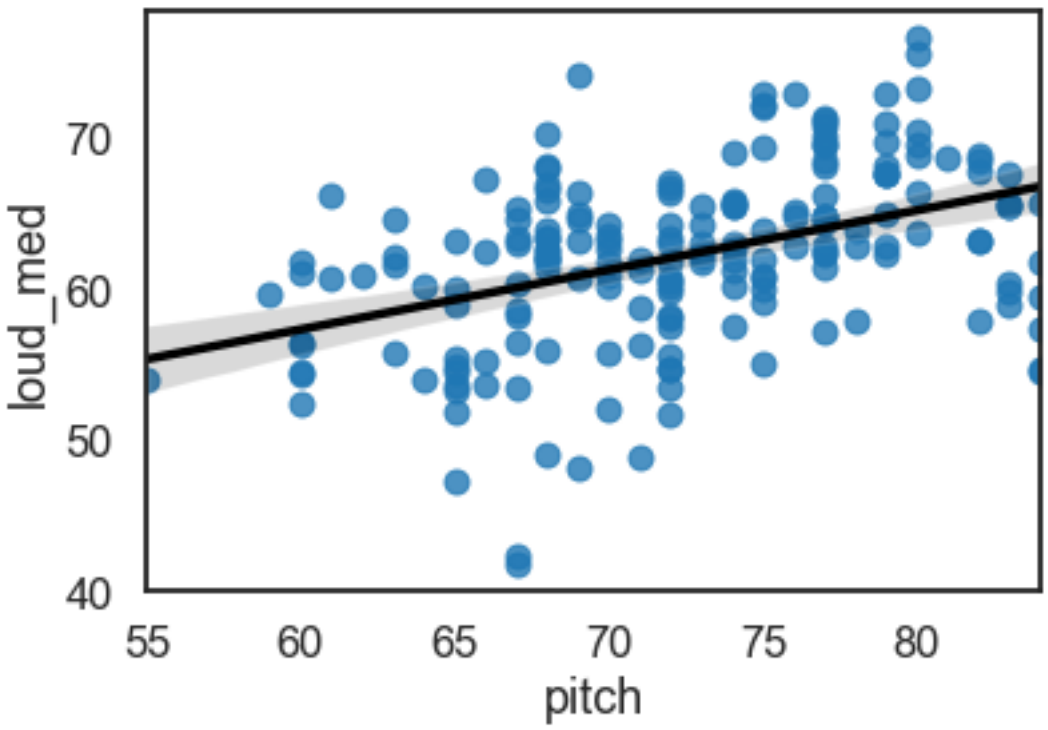
We can get a visual impression of whether there might be a direct relation between the two features by plotting it and drawing a regression line. For this, the `regplot()` function of the `seaborn` library is well-suited.

```
sns.regplot(data=example_solo, x="pitch", y="loud_med", line_kws={"color":"black"});
```



There seems to be no clear relation; no matter how high the pitch, the loudness stays more or less the same. Let's look at another example!

```
example_solo2 = solos[ solos["melid"] == 333 ][["pitch", "loud_med"]]

sns.regplot(data=example_solo2, x="pitch", y="loud_med", line_kws={"color":"black"});
```



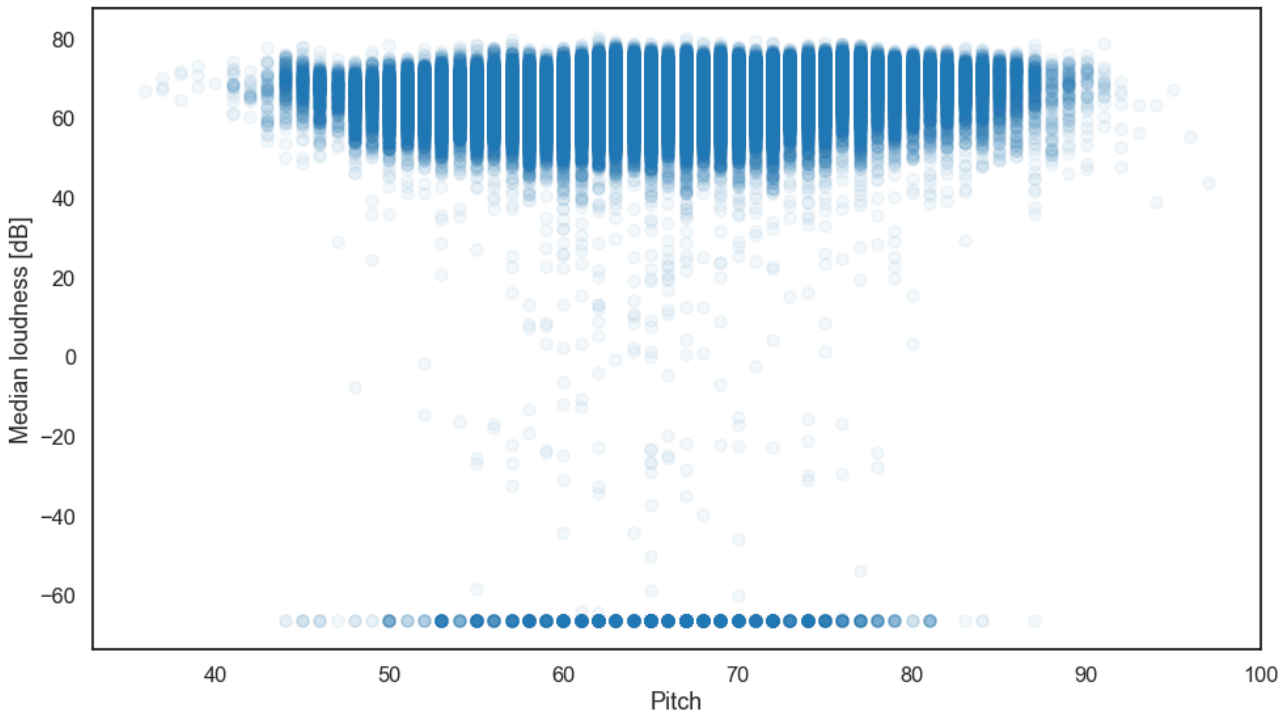
In this case, there is a positive trend. The higher the pitch, the louder the performer plays. Since we

have now two different examples - in one case no relation, in the other case a positive correlation - we should now look at whether there is a trend emerging from all solos taken together.

14.3. The "rain cloud" of Jazz solos

We now take all 200'809 notes from all solos and look at the relation between their pitch and their median loudness.

```
X = solos[["pitch", "loud_med"]].values
x = X[:,0]
y = X[:,1]

fig, ax = plt.subplots(figsize=(16,9))
ax.scatter(x,y, alpha=0.05)

plt.xlabel("Pitch")
plt.ylabel("Median loudness [dB]")
plt.show()
```



The visual impression is that of a cloud from which rain drops down and forms a puddle. Which trends can we observe?

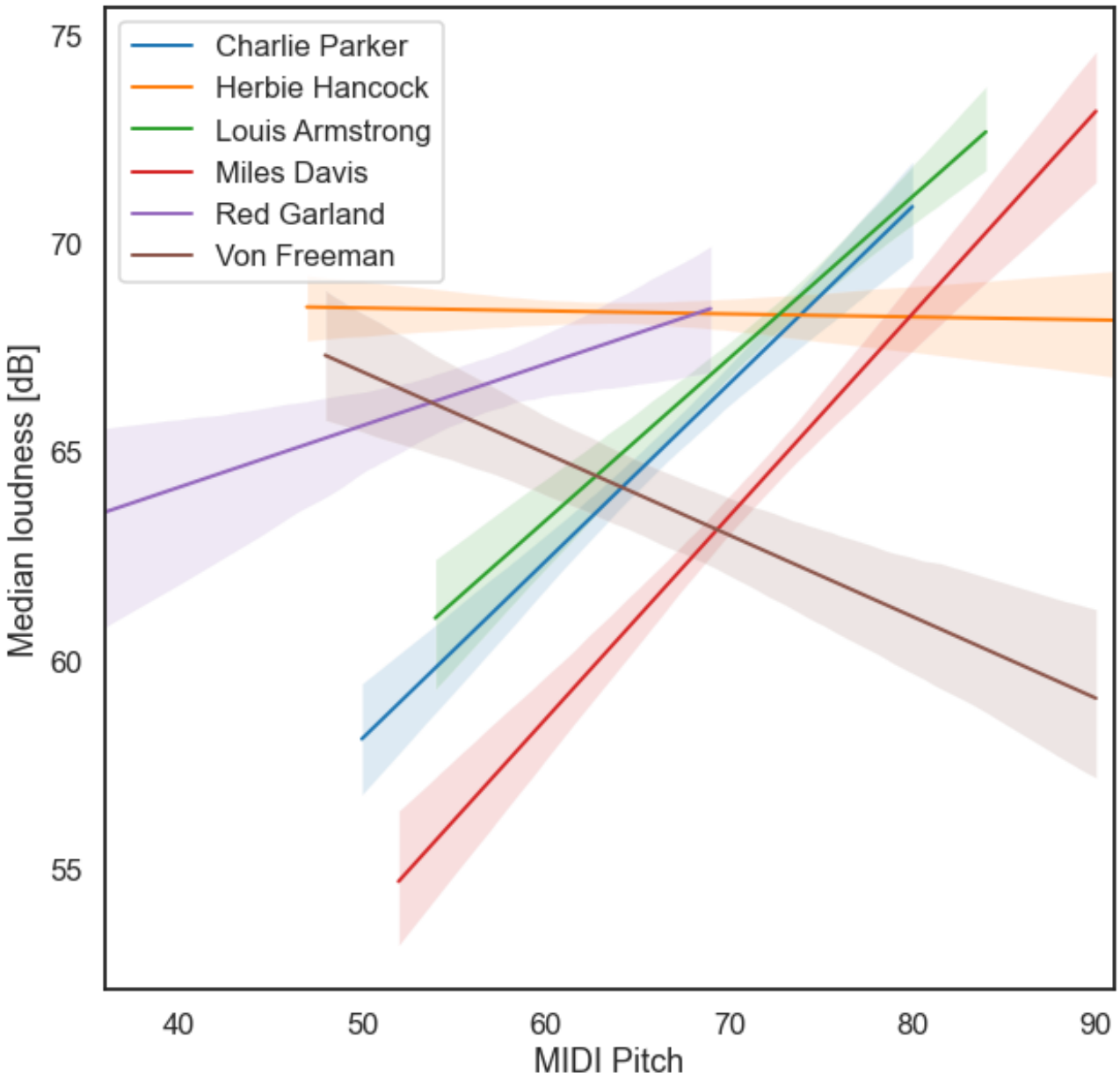
14.4. Comparing performers

Taking all pieces together was not really informative. Maybe a somewhat closer look brings more to the front. Let us some specific performers whose solos we want to compare.

```
selected_performers = ["Charlie Parker", "Miles Davis", "Louis Armstrong", "Herbie Hancock", "
```

```
grouped_df = solos.groupby("performer")
```

```
fig, ax = plt.subplots(figsize=(10,10))

for performer, df in grouped_df:
    if performer in selected_performers:
        sns.regplot(
            data=df,
            x="pitch",
            y="loud_med",
            x_jitter=.1,
            y_jitter=.1,
            scatter_kws={"alpha":.01, "color":"grey"},
            line_kws={"lw":2},
            label=performer,
            scatter=False,
            ax=ax
        )

plt.xlabel("MIDI Pitch")
plt.ylabel("Median loudness [dB]")
plt.legend()
plt.show()
```



Observations:

1. Most performers increase loudness with increasing pitch.
2. Charlie Parker (sax) and Louis Armstrong (t) show very similar patterns but Armstrong is generally higher.
3. Miles Davis (t) is similar to the two but plays generally softer than both.
4. Von Freeman (sax) strongly and Herbie Hancock (p) weakly decrease loudness with increasing pitch (almost all other performers show positive correlations).
5. Red Garland (p) plays generally lower than Herbie Hancock (p) but does show a positive correlation between pitch and loudness (NB: there is only one solo in the database).

Does this tell us something about performer styles or about instruments?

15. Data-Driven Music History

```
import pandas as pd # for working with tabular data
pd.set_option('display.max_columns', 500)
import matplotlib.pyplot as plt # for plotting
plt.style.use("fivethirtyeight") # select specific plotting style
import seaborn as sns; sns.set_context("talk")
import numpy as np
```

15.1. Research Questions

- General: How can we study historical changes quantitatively?
- Specific: What can we say about the history of tonality based on a dataset of musical pieces?

15.2. A bit of theory

```
note_names = list("FCGDAEB") # diatonic note names in fifths ordering
note_names
```

```
['F', 'C', 'G', 'D', 'A', 'E', 'B']
```

```
accidentals = ["bb", "b", "", "#", "##"] # up to two accidentals is suffient here
accidentals
```

```
['bb', 'b', '', '#', '##']
```

```
lof = [ n + a for a in accidentals for n in note_names ] # lof = "Line of Fifths"
print(lof)
```

```
['Fbb', 'Cbb', 'Gbb', 'Dbb', 'Abb', 'Ebb', 'Bbb', 'Fb', 'Cb', 'Gb', 'Db', 'Ab', 'Eb', 'Bb', 'F
```

```
len(lof) # how long is this line-of-fifths segment?
```

```
35
```

We call the elements on the line of fifths **tonal pitch-classes**

15.3. Data

15.3.1. A (kind of) large corpus: TP3C

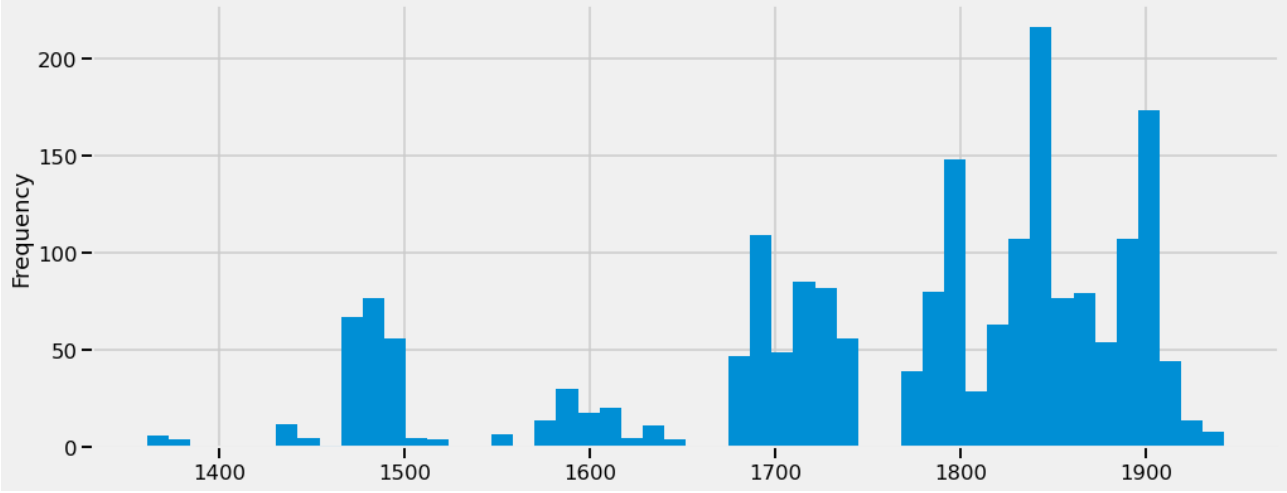
Here, we use a dataset that was specifically compiled for this kind of analysis, the **Tonal pitch-class counts corpus (TP3C)** (Moss, Neuwirth, Rohrmeier, 2020)

- 2,012 pieces
- 75 composers
- approx. spans 600 years of music history
- does not contain complete pieces but only counts of tonal pitch-classes

```
import pandas as pd # to work with tabular data

url = "https://raw.githubusercontent.com/DCMLab/TP3C/master/tp3c.tsv"
data = pd.read_table(url)

data.sample(10)
```

	composer	composer_first	work_group	work_catalogue	opus	no	mov	tit
1742	Corelli	Arcangelo	12 Trio Sonatas	Op.	3	4	1.0	Na
468	Bach	Johann Sebastian	Inventions and Sinfonias	BWV	779	NaN	NaN	Na
1630	Chopin	Frédéric	Mazurkas	Op.	33	2	NaN	Na
1604	Joplin	Scott	Ragtimes	NaN	NaN	NaN	NaN	Lil
137	Bach	Johann Sebastian	Wohltemperiertes Klavier II	BWV	888	2	NaN	Na
1231	Schubert	Franz	Die schöne Müllerin	D. 795	NaN	18	NaN	Tr
530	Brahms	Johannes	8 Klavierstücke	Op.	76	4	NaN	Int
1311	Schumann	Robert	Dichterliebe	Op.	48	3	NaN	Di
338	Bach	Johann Sebastian	Wohltemperiertes Klavier I	BWV	863	1	NaN	Na
713	Fauré	Gabriel	NaN	Op.	6	2	NaN	Tri

```
data["display_year"].plot(kind="hist", bins=50, figsize=(15,6)); # historical overview
```



- it can be seen that there are large gaps and that some historical periods are underrepresented
- however, it is not so obvious how to fix that

- do we want a uniform distribution over time?
- do we want a "historically accurate" distribution?
- do we want to remove geographical/gender/class/instrument/etc. biases?
- on one hand, balanced datasets are likely not to reflect historical realities
- on the other hand, such datasets rather represent the "canon", that is a contemporary selection of "valuable" compositions that may differ greatly from what was considered relevant at the time

–> There is no unique objective answer to these questions. It is important to be aware of these limitations and take them into account when interpreting the results

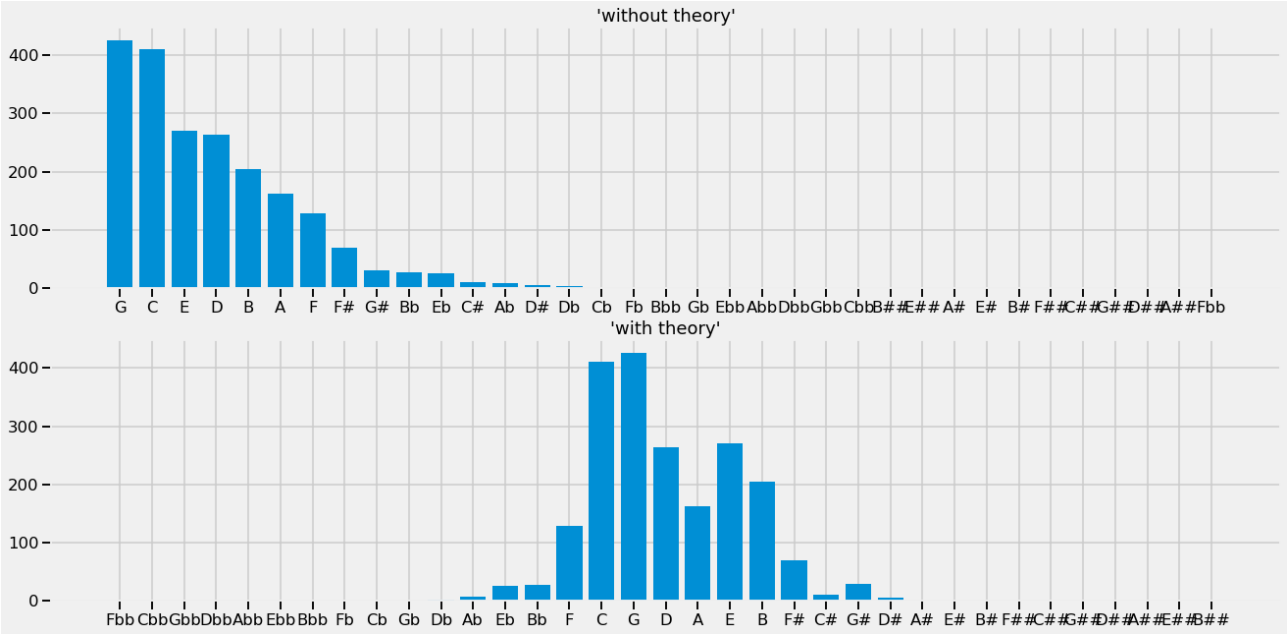
For this workshop we ignore all the metadata about the pieces (titles, composer names etc.) but only focus on their tonal material. Therefore, we don't need all the columns of the table.

```
tpc_counts = data.loc[:, lof] # select all rows (":") and the lof columns
tpc_counts.sample(20)
```

	Fbb	Cbb	Gbb	Dbb	Abb	Ebb	Bbb	Fb	Cb	Gb	Db	Ab	Eb	Bb	F	C	G
155	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1073	0	0	0	0	0	0	0	0	11	23	83	189	219	114	229	225	323
1975	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1178	0	0	0	0	0	0	0	0	0	0	0	0	11	84	101	77	111
402	0	0	0	0	0	0	0	0	1	2	1	2	2	9	13	100	214
98	0	0	0	0	2	0	4	2	6	41	118	144	114	164	241	340	142
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	52	193	249
344	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	2	6
1784	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
407	0	0	0	0	0	0	0	0	0	0	0	3	0	11	148	180	168
1992	0	0	0	0	0	0	0	0	3	4	27	138	315	155	253	327	490
543	0	0	0	0	0	0	0	0	0	17	3	0	28	61	36	52	120
1046	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7	11
1032	0	0	0	0	0	1	5	4	18	35	47	132	95	205	173	155	347
737	0	0	0	0	19	22	16	41	103	101	33	48	251	183	97	31	9
186	0	0	0	0	0	0	0	0	0	0	4	2	16	97	176	198	138
1653	0	0	0	0	0	0	0	0	0	0	1	2	0	16	123	237	223
1115	0	0	0	0	0	0	0	0	0	10	8	6	41	64	42	20	48
1886	0	0	0	0	0	0	0	0	0	0	6	149	233	174	230	269	408
343	0	0	0	0	0	0	0	0	0	0	0	1	19	53	102	146	102

```
piece = tpc_counts.iloc[10]

fig, axes = plt.subplots(2, 1, figsize=(20,10))

axes[0].bar(piece.sort_values(ascending=False).index, piece.sort_values(ascending=False))
axes[0].set_title("'without theory'")

axes[1].bar(piece.index, piece)
axes[1].set_title("'with theory'")

# plt.savefig("img/random_piece.png")
# plt.show()
```

```
Text(0.5, 1.0, "'with theory'")
```



Let us have an overview of the note counts in these pieces!

If we would just look at the raw counts of the tonal pitch-classe, we could not learn much from it. Using a theoretical model (the line of fifths) shows that the notes in pieces are usually come from few adjacent keys (you don't say!).

We probably have very long pieces (sonatas) and very short pieces (songs) in the dataset. Since we don't want length (or the absolute number of notes in a piece) to have an effect, we rather consider tonal pitch-class distributions instead counts, by normalizing all pieces to sum to one.

```
tpc_dists = tpc_counts.div(tpc_counts.sum(axis=1), axis=0)
tpc_dists.sample(20)
```

	Fbb	Cbb	Gbb	Dbb	Abb	Ebb	Bbb	Fb	Cb	Gb	Db	A
606	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0
1844	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0
251	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0
737	0.0	0.0	0.0	0.0	0.017288	0.020018	0.014559	0.037307	0.093722	0.091902	0.030027	0
10	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.001466	0
822	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0
202	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.003268	0.011438	0
850	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0
1609	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0
555	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0
940	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.005703	0.002852	0.004753	0.046578	0
1165	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0
1438	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0
1548	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.011111	0
1265	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.005924	0.000000	0.004739	0
1169	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.011472	0.024857	0

	Fbb	Cbb	Gbb	Dbb	Abb	Ebb	Bbb	Fb	Cb	Gb	Db	A
1368	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0
1741	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0
1302	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.002725	0.013624	0
80	0.0	0.0	0.0	0.0	0.001965	0.002947	0.000000	0.008841	0.030452	0.051081	0.090373	0

For further numerical analysis, we extract the data from this table and assign it to a variable `X`.

```
# extract values of table to matrix
X = tpc_dists.values

X.shape # shows (#rows, #columns) of X
```

```
(2012, 35)
```

Now, `X` is a 2012 × 35 matrix where the rows represent the pieces and the columns (also called "features" or "dimensions") represent the relative frequency of tonal pitch-classes.

Thinking in 35 dimensions is quite difficult for most people. Without trying to imagine what this would look like, what can we already say about this data?

Since each piece is a point in this 35-D space and pieces are represented as vectors, pieces that have similar tonal pitch-class distributions must be close in this space (whatever this looks like).

What groups of pieces that cluster together? Maybe pieces of the same composer are similar to each other? Maybe pieces from a similar time? Maybe pieces for the same instruments?

If we find clusters, these would still be in 35-D and thus difficult to interpret. Luckily, there are a range of so-called *dimensionality reduction* methods that transform the data into lower-dimensional spaces so that we actually can look at them.

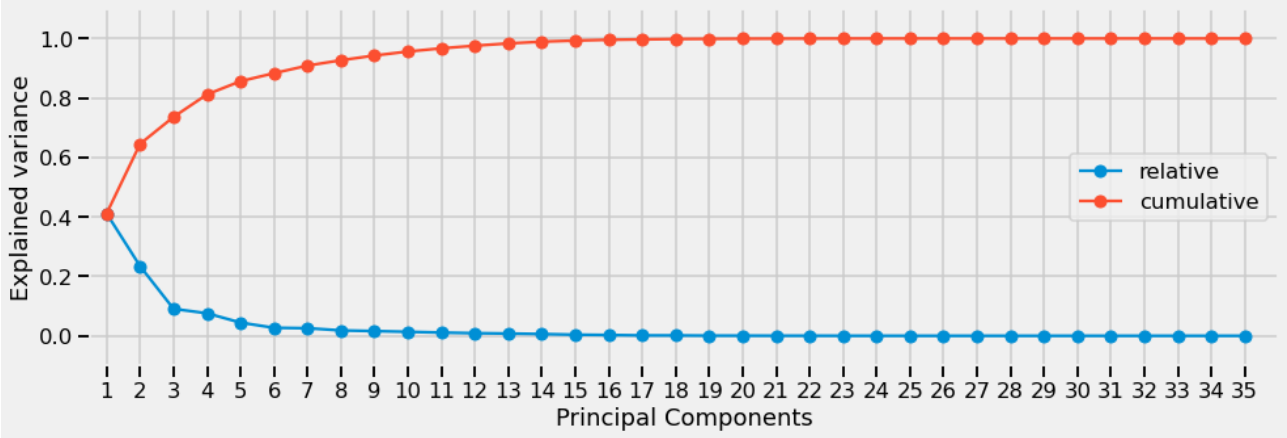
A very common dimensionality reduction method is **Principal Components Analysis (PCA)**.

The basic idea of PCA is:

- find dimensions in the data that maximize the variance in this direction
- these dimensions have to be orthogonal to each other (mutually independent)
- these dimensions are called the *principal components*
- each principal component is associated with how much of the data variance it explains

```
import numpy as np # for numerical computations
import sklearn
from sklearn.decomposition import PCA # for dimensionality reduction

pca = sklearn.decomposition.PCA(n_components=35) # initialize PCA with 35 dimensions
pca.fit(X) # apply it to the data
variance = pca.explained_variance_ratio_ # assign explained variance to variable
```

```
fig, ax = plt.subplots(figsize=(14,5))
x = np.arange(35)
ax.plot(x, variance, label="relative", marker="o")
ax.plot(x, variance.cumsum(), label="cumulative", marker="o")
ax.set_xlim(-0.5, 35)
ax.set_ylim(-0.1, 1.1)
ax.set_xlabel("Principal Components")
ax.set_ylabel("Explained variance")
plt.xticks(np.arange(len(lof)), np.arange(len(lof)) + 1) # because Pyhon starts counting at 0

plt.legend(loc="center right")
plt.tight_layout()
# plt.savefig("img/explained_variance.png")
# plt.show()
```




```
variance[:5]
```

```
array([0.41144591, 0.23410347, 0.09063507, 0.07574242, 0.04436989])
```

The first principal component explains 41.1% of the variance of the data, the second explains 23.4% and the third 9%. Together, this amounts to 73.6%.

Almost three quarters of the variance in the dataset is retained by reducing the dimensionality from 35 to 3 dimensions (8.6%)! If we reduce the data to two dimensions, we still can explain ≈ 65% of the variance.

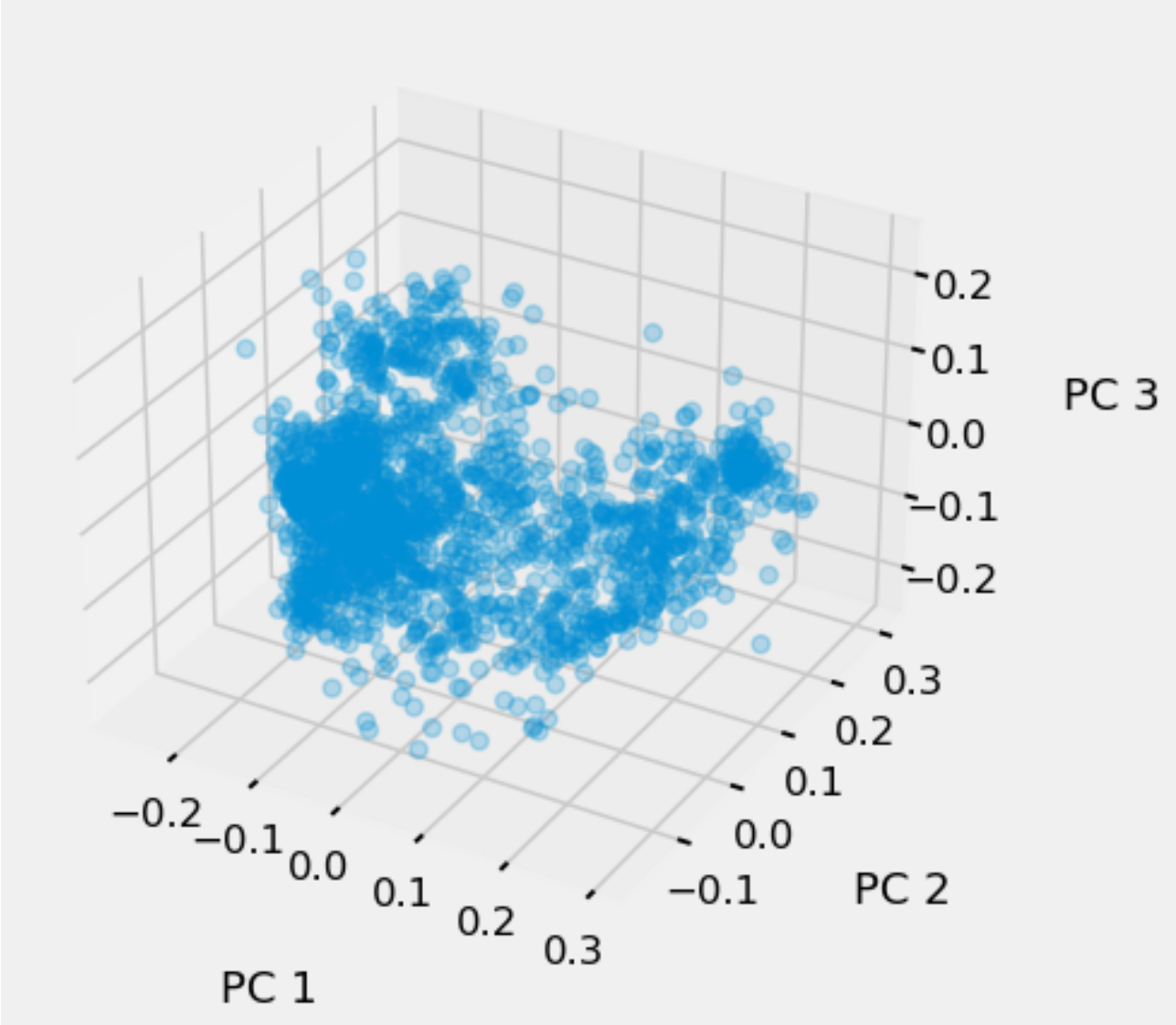
This is great because it means that we can look at the data in 2 or 3 dimensions without loosing too much information.

15.4. Recovering the line of fifths from data

```
pca3d = PCA(n_components=3)
pca3d.fit(X)

X_ = pca3d.transform(X)
X_.shape
```

```
(2012, 3)
```

```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(6,6))

ax = fig.add_subplot(111, projection='3d')
ax.scatter(X_[:,0], X_[:,1], X_[:,2], s=50, alpha=.25) # c=cs,
ax.set_xlabel("PC 1", labelpad=30)
ax.set_ylabel("PC 2", labelpad=30)
ax.set_zlabel("PC 3", labelpad=30)

plt.tight_layout()
# plt.savefig("img/3d_scatter.png")
# plt.show()
```

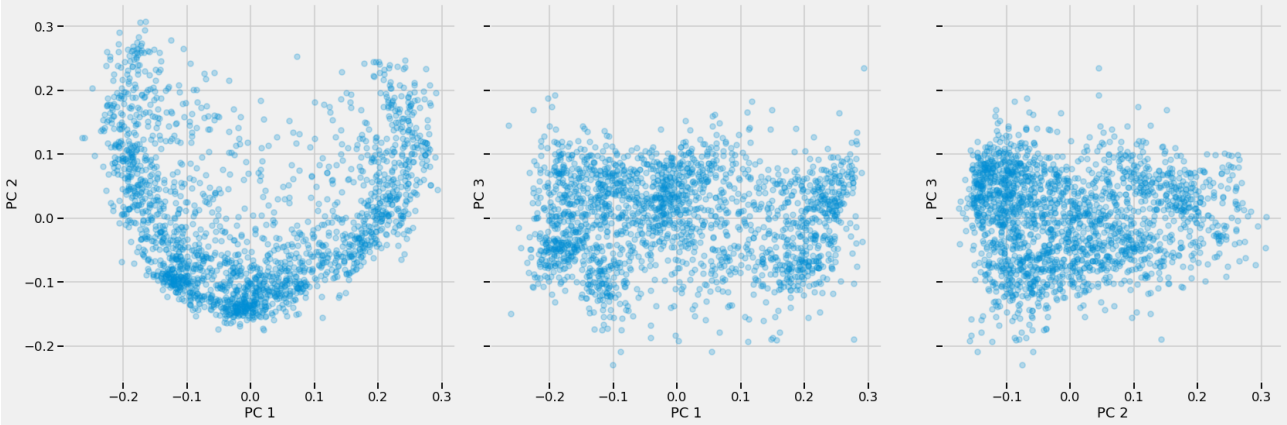


Each piece in this plot is represented by a point in 3-D space. But remember that this location represents ~75% of the information contained in the full tonal pitch-class distribution. In 35-D space each dimension corresponded to the relative frequency of a tonal pitch-class in a piece.

- What do these three dimensions signify?
- How can we interpret them?

Fortunately, we can inspect them individually and try to interpret what we see.

```
from itertools import combinations

fig, axes = plt.subplots(1,3, sharey=True, figsize=(24,8))

for k, (i, j) in enumerate(combinations(range(3), 2)):

    axes[k].scatter(X_[:,i], X_[:,j], s=50, alpha=.25, edgecolor=None)
    axes[k].set_xlabel(f"PC {i+1}")
    axes[k].set_ylabel(f"PC {j+1}")
    axes[k].set_aspect("equal")

plt.tight_layout()
# plt.savefig("img/3d_dimension_pairs.png")
# plt.show()
```



Clearly, looking at two principal components at a time shows that there is some latent structure in the data. How can we understand it better?

One way to see whether the pieces are clustered together systematically be coloring them according to some criterion.

As always, many different options are available. For the present purpose we will use the most simple summary of the piece: its most frequent note (which is the *mode* of its pitch-class distribution in statistical terms) and call this note its **tonal center**.

This will also allow to map the tonal pitch-classes on the line of fifths to colors.

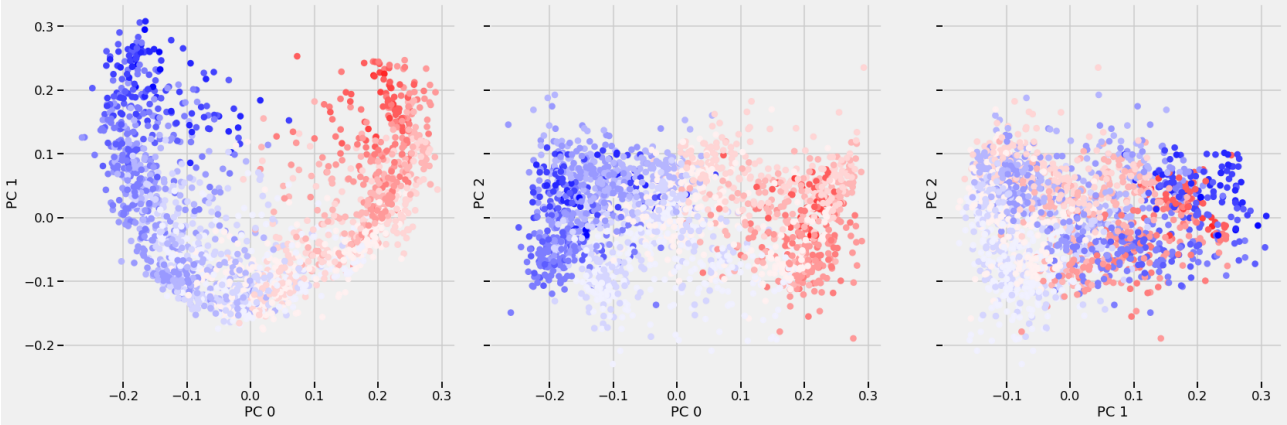
```
tpc_dists["tonal_center"] = tpc_dists.apply(lambda piece: np.argmax(piece[lof].values) - 15, a
tpc_dists.sample(10)
```

	Fbb	Cbb	Gbb	Dbb	Abb	Ebb	Bbb	Fb	Cb	Gb	Db	Ab	Eb
1990	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.002752	0.004587	0.015138	0.088991	0.13
364	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.001026	0.006028	0.020521	0.037194	0.03
1857	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.000000	0.000000	0.000000	0.007370	0.03

	Fbb	Cbb	Gbb	Dbb	Abb	Ebb	Bbb	Fb	Cb	Gb	Db	Ab	Eb
1045	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.000000	0.000000	0.000000	0.000000	0.000
1246	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01232	0.020534	0.008214	0.131417	0.185832	0.320
214	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.000000	0.000000	0.000000	0.000000	0.000
167	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.000000	0.000000	0.000000	0.000000	0.000
570	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.000000	0.000000	0.000000	0.000000	0.000
586	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.000000	0.000000	0.001506	0.006274	0.041
684	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.000000	0.000000	0.000000	0.000000	0.000

```
from matplotlib import cm
from matplotlib.colors import Normalize

#normalize item number values to colormap
norm = Normalize(vmin=-15, vmax=20)

# cs = [ cm.seismic(norm(c)) for c in data["tonal_center"]]
cs = [ cm.seismic(norm(c)) for c in tpc_dists["tonal_center"]]
```

```
from itertools import combinations

fig, axes = plt.subplots(1,3, sharey=True, figsize=(24,8))

for k, (i, j) in enumerate(combinations(range(3), 2)):

    axes[k].scatter(X_[:,i], X_[:,j], s=50, c=[ np.abs(c) for c in cs], edgecolor=None)
    axes[k].set_xlabel(f"PC {i}")
    axes[k].set_ylabel(f"PC {j}")
    axes[k].set_aspect("equal")

plt.tight_layout()
# plt.savefig("img/3d_dimension_pairs_colored.png")
# plt.show()
```



15.5. Historical development of tonality

The line of fifths is an important underlying structure for pitch-class distributions in tonal compositions

But we have treated all pieces in our dataset as synchronic and have not yet taken their historical location into account.

Let's assume the pitch-class content of a piece spreads on the line of fifths from F to A♯. This means, its range on the line of fifths is $10 - (-1) = 11$. The piece covers eleven consecutive fifths on the lof.

We can generalize this calculation and write a function that calculates the range for each piece in the dataset.

```
def lof_range(piece):
    l = [i for i, v in enumerate(piece) if v!=0]
    return max(l) - min(l)
```

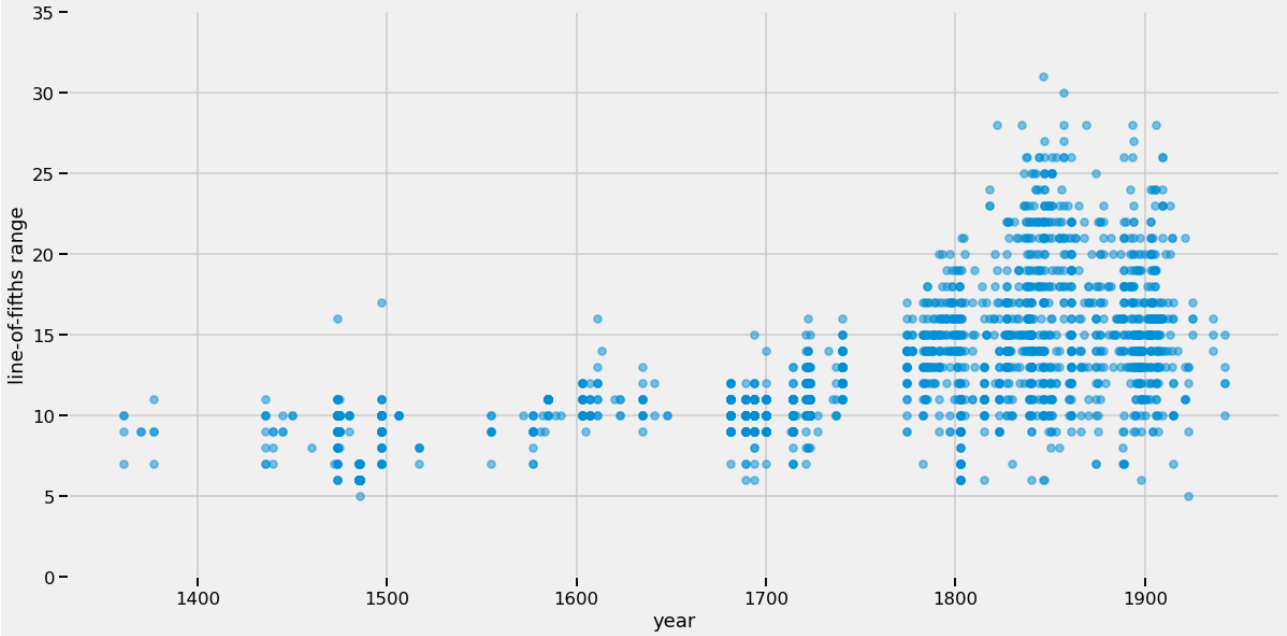
```
data["lof_range"] = data.loc[:, lof].apply(lof_range, axis=1) # create a new column
data.sample(20)
```

	composer	composer_first	work_group	work_catalogue	opus	no
992	Agricola	Alexander	Missa Malheur me bat	NaN	NaN	NaN
1772	Corelli	Arcangelo	12 Trio Sonatas	Op.	4	1
1199	Scarlatti	Domenico	Sonata	K	64	NaN
1361	Scriabin	Alexander	Préludes	Op.	13	2
1490	Tchaikovsky	Pyotr	The Seasons	Op.	37a	3
387	Chopin	Frédéric	Préludes	Op.	28	6
290	Alkan	Charles Valentin	Préludes	Op.	31	13
478	Bach	Johann Sebastian	Inventions and Sinfonias	BWV	789	NaN
1947	Mozart	Wolfgang Amadeus	Sonaten	KV	283	5
601	Couperin	François	Troisième livre de pièces de Clavecin	NaN	NaN	18
974	Ockeghem	JeanDe	Missa Fors Seulement	NaN	NaN	NaN
1430	Victoria	TomasLuisde	NaN	NaN	NaN	NaN
528	Brahms	Johannes	8 Klavierstücke	Op.	76	2
1765	Corelli	Arcangelo	12 Trio Sonatas	Op.	3	8
1300	Schumann	Clara	Sechs Lieder	Op.	23	5
204	Liszt	Franz	12 Transcendental Etudes	S.	139	3
1858	Grieg	Edvard	Lyrical Pieces	Op.	65	6
468	Bach	Johann Sebastian	Inventions and Sinfonias	BWV	779	NaN
439	Victoria	TomasLuisde	NaN	NaN	NaN	NaN
441	Alkan	Charles Valentin	NaN	Op.	25	NaN

This allows us now to take the `display_year` (composition or publication) and `lof_range` (range on the line of fifths) features to observe historical changes.

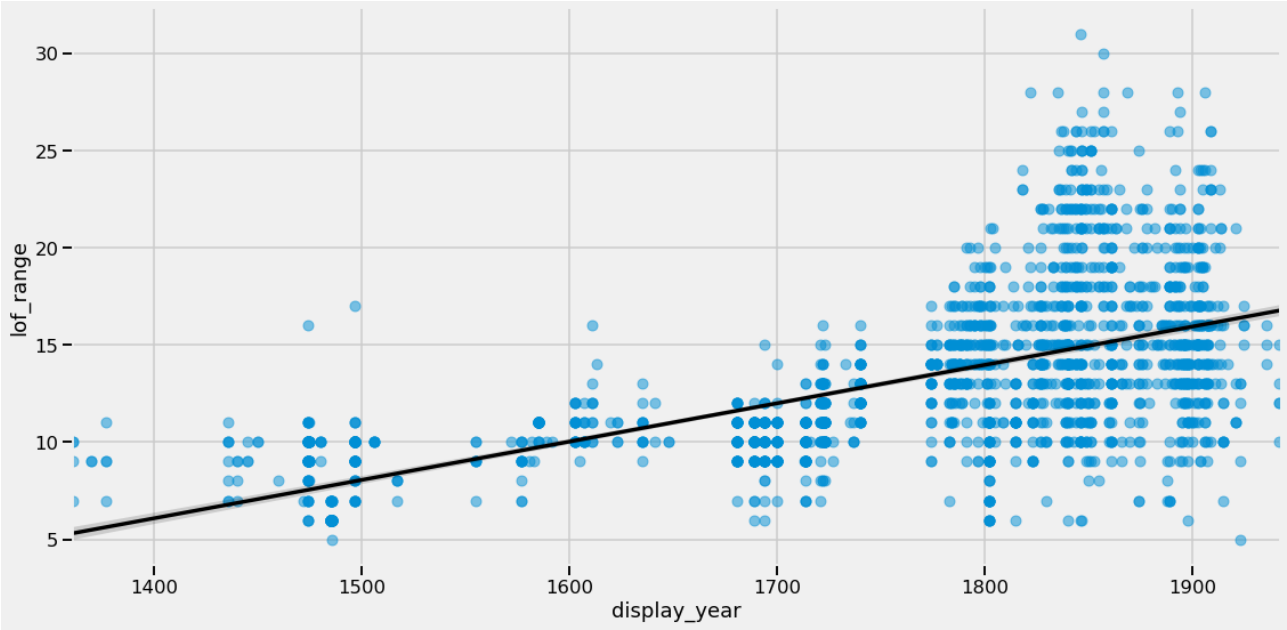
```
fig, ax = plt.subplots(figsize=(18,9))
ax.scatter(data["display_year"].values, data["lof_range"].values, alpha=.5, s=50)
ax.set_ylim(0,35)
ax.set_xlabel("year")
```

```
ax.set_ylabel("line-of-fifths range");
# plt.savefig("img/hist_scatter.png");
```



We could try to fit a line to this data to see whether there is a trend (kinda obvious here).

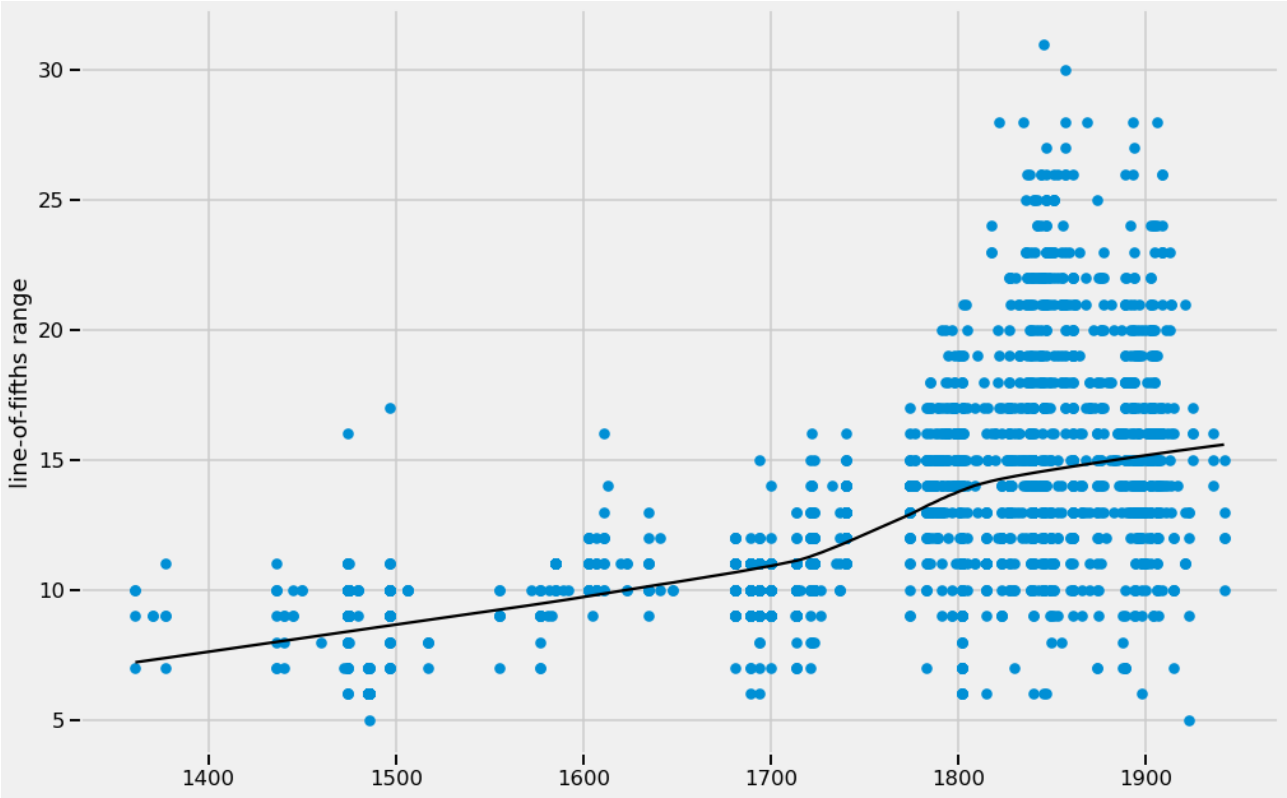
```
g = sns.lmplot(
    data=data,
    x="display_year",
    y="lof_range",
    line_kws={"color":"k"},
    scatter_kws={"alpha":.5},
#     lowess=True,
    height=8,
    aspect=2
);
# g.savefig("img/hist_scatter_line.png");
```



But actually, this is not the best idea. Why should any historical process be linear? More complex models might make more sense.

A more versatile technique is *Locally Weighted Scatterplot Smoothing* (LOWESS) that locally fits a polynomial. Using this method, we see that a non-linear process is displayed.

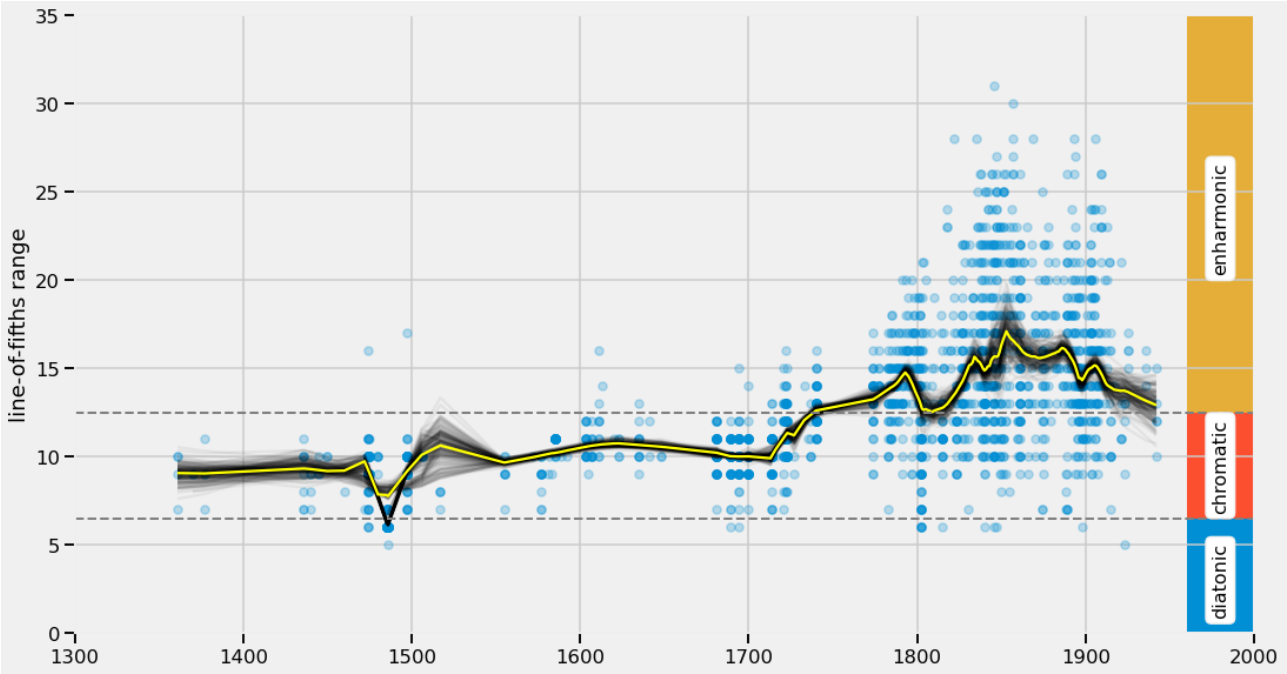
```
from statsmodels.nonparametric.smoothers_lowess import lowess

x = data.display_year
y = data.lof_range
l = lowess(y,x)

fig, ax = plt.subplots(figsize=(15,10))

ax.scatter(x,y, s=50)
ax.plot(l[:,0], l[:,1], c="k")
ax.set_ylabel("line-of-fifths range");
# plt.savefig("img/hist_scatter_lowess.png")
# plt.show()
```



15.6. If there is time: some more advanced stuff

```
B = 200
delta = 1/10

fig, ax = plt.subplots(figsize=(16,9))

x = data.display_year
y = data.lof_range
l = lowess(y,x, frac=delta)

ax.scatter(x,y, s=50, alpha=.25)

for _ in range(B):
    resampled = data.sample(data.shape[0], replace=True)

    xx = resampled.display_year
    yy = resampled.lof_range
    ll = lowess(yy,xx, frac=delta)

    ax.plot(ll[:,0], ll[:,1], c="k", alpha=.05)

ax.plot(l[:,0], l[:,1], c="yellow")

## REGIONS
from matplotlib.patches import Rectangle
```

```
text_kws = {
    "rotation" : 90,
    "fontsize" : 16,
    "bbox" : dict(
        facecolor="white",
        boxstyle="round"
    ),
    "horizontalalignment" : "center",
    "verticalalignment" : "center"
}

rect_props = {
    "width" : 40,
    "zorder" : -1,
    "alpha" : 1.
}

stylecolors = plt.rcParams["axes.prop_cycle"].by_key()["color"]

ax.text(1980, 3, "diatonic", **text_kws)
ax.axhline(6.5, c="gray", linestyle="--", lw=2) # dia / chrom.
ax.add_patch(Rectangle((1960,0), height=6.5, facecolor=stylecolors[0], **rect_props))

ax.text(1980, 9.5, "chromatic", **text_kws)
ax.axhline(12.5, c="gray", linestyle="--", lw=2) # chr. / enh.
ax.add_patch(Rectangle((1960,6.5), height=6, facecolor=stylecolors[1], **rect_props))

ax.text(1980, 23.5, "enharmonic", **text_kws)
ax.add_patch(Rectangle((1960,12.5), height=28, facecolor=stylecolors[2], **rect_props))

ax.set_ylim(0,35)
ax.set_xlim(1300,2000)

ax.set_ylabel("line-of-fifths range");
# plt.savefig("img/final.png", dpi=300)
# plt.show()
```



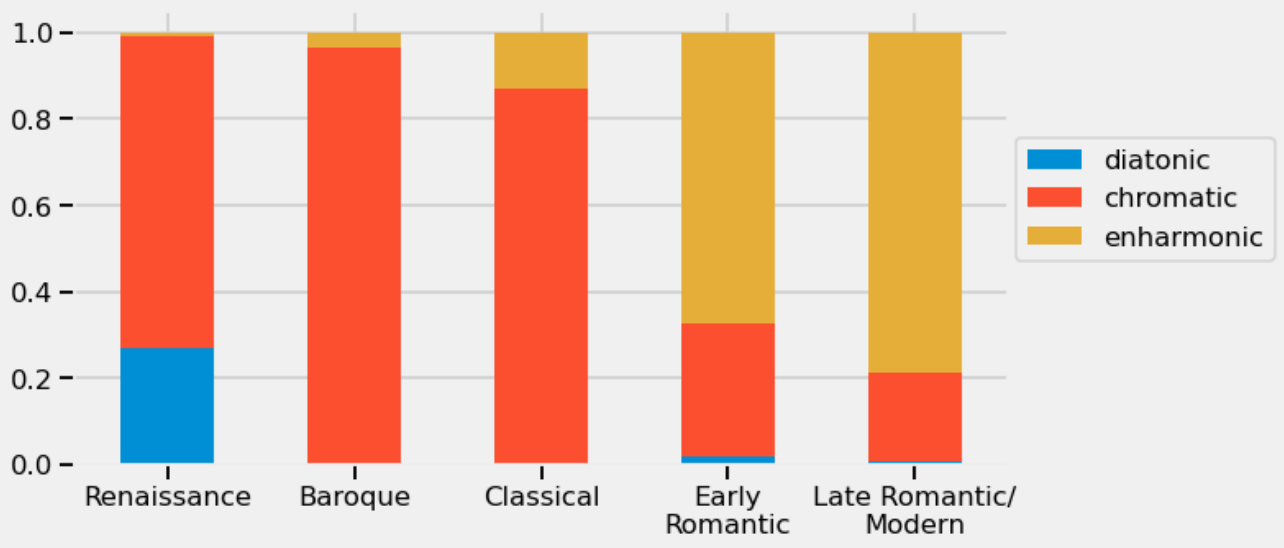
Usung bootstrap sampling we achieve an estimation of the local varience of the data and thus of the diversity in the note usage of the musical pieces.

We also can distinguish three regions in terms of line-of-fifth range: diatonic, chromatic, and enharmonic.

Grouping the data together in these three regions, we see a clear change from diatonic and chromatic to chromatic and enharmonic pieces over the course of history.

```
epochs = {
    "Renaissance" : [1300, 1549],
    "Baroque" : [1550, 1649],
    "Classical" : [1650, 1749],
    "Early\nRomantic" : [1750, 1819],
    "Late Romantic/\nModern" : [1820, 2000]
}

strata = [
    "diatonic",
    "chromatic",
    "enharmonic"
]

widths = data[["display_year", "lof_range"]].sort_values(by="display_year").reset_index(drop=T

df = pd.concat(
    [
        widths[
            (widths.display_year >= epochs[e][0]) & (widths.display_year <= epochs[e][1])
        ]["lof_range"].value_counts(normalize=True).sort_index().groupby(
            lambda x: strata[0] if x <= 6 else strata[1] if x <= 12 else strata[2]
        ).sum() for e in epochs
    ], axis=1, sort=True
```

```
)

df.columns = epochs.keys()
df = df.reindex(strata)
df.T.plot(kind="bar", stacked=True, figsize=(12,5))
# plt.title("Epochs")
plt.legend(bbox_to_anchor=(1.3,0.75))
plt.gca().set_xticklabels(epochs.keys(), rotation="horizontal")
plt.tight_layout()
# plt.savefig("img/epochs_regions.png")
plt.show()
```



- Renaissance: largest diatonic proportion overall but mostly chromatic
- Baroque: alost completely chromatic
- Classical: enharmonic proportion increases -> more distant modulations
- This trend continues through the Romantic eras

15.7. Summary

1. We have analyzed a very specific aspect of Western classical music.
2. We have used a large(-ish) corpus to answer our research question.
3. We have operationalized musical pieces as vectors that represent distributions of tonal pitch-classes.
4. We have used the dimensionality-reduction technique Principal Component Analysis (PCA) in order to visually inspect the distribution of the data in 2 and 3 dimensions.
5. We have used music-theoretical domain knowledge to find meaningful structure in this space.
6. We have seen that pieces are largely distributed along the line of fifths.
7. We have used Locally Weighted Scatterplot Smoothing (LOWESS) to estimate the variance in this historical process.
8. We have seen that, historically, composers explore ever larger regions on this line and that the variance also increases.

Part V.

CRITICAL DIGITAL MUSICOLOGY

16. Copyright

> **Goal**
>
> Know a few famous copyright infringement cases and why data analysis is important here.

- Plagiarism cases and copyright

17. Representation and representativeness

> **Goal**
>
> Understand the difference between representativeness and representation. Obtain a critical understanding of biases relevant for data selection.

- Representation and the canon
- Representing means modeling means abstraction (what is “music” in “music encoding”?)
- biases: how to recognize them, how to deal with them, and when biases are a good thing.
- FAIR and CARE

18. Discussion

> Goal

Part VI.

EXERCISES

Exercise for Week 1

In the remainder of the course, we will engage with data and computer code. While this is probably new for most of you, there are luckily many tools that can help us to do so more easily.

1. Create a directory/folder named `intro-digimus` for this course (**note:** no spaces!). You can use any location on your personal computer or on your J: drive provided by the university.
2. As code editor we will use Microsoft Visual Studio Code. Install the program on your machine.
3. Open the directory you just created and install the Jupyter Extension.
4. Create a new Jupyter notebook named `test.ipynb` and write the following in the first cell: `print("Hello world!")`. Congratulations, you wrote your first computer program!

Homework

1. Read Schaffer (2016).
2. Discuss with your peers unclear terms and find answers.
3. Find aspects of Computational Musicology that you think are underrepresented in Schaffer's list.

19. A Python primer

> **Note**
>
> From now on, we will assume that you have a working Python installation running on your computer. You can check this by typing the following into a terminal/console/command line:
>
> ```
> python --version
> ```
>
> If the version number starts with a 3, you're all set. If not, please consider one of the many tutorials online on how to install Python.

19.1. Variables and types

Variable assignment in Python is straight-forward. You choose a name for the variable, and assign a value to it using the `=` operator, for example:

```
x = 100
```

assigns the value `100` to the variable `x`. If we call the variable now, we can see that it has the value we assigned to it:

```
x
```

```
100
```

Of course, we can also assign things other than numbers, for example:

```
name = "Fabian"
```

What we assigned to the variable `name` is called a *string*, it has the value `"Fabian"`. Strings are sequences of characters.

> **Tip**
>
> Note that `"Fabian"` is enclosed by double-quotes. Why is this the case? Why could we not just type `name = Fabian`?

We can also assign a list of things to a variable:

```
mylist = [1, 2, 3, "A", "B", "C"]
```

Lists are enclosed by square brackets. As you can see, Python allows you to store any kind of data in lists (here: integer numbers and character strings). However, it is good practice to include only items of the same type in lists—you'll understand later why.

Another structured data type in python are dictionaries. Dictionaries are collections of key-value pairs. For example, a dictionary `addresses` could store the email addresses of certain people:

```
addresses = {
    "Andrew" : "andrew@example.com",
    "Zoe" : "zoe@example.com"
}
```

Now, if we wanted to look up Zoe's email address, we could to so with:

```
addresses["Zoe"]
```

```
'zoe@example.com'
```

19.2. On repeat

Coding something is only useful if you can't do the job as fast or as efficient by yourself. Especially when it comes to repeating the same little thing many, many times, knowing how to code comes in handy.

As a simple example, imagine you want to write down all numbers from 1 to 10, or from 1 to 100, or… you get the idea. In Python, you would do it as follows:

```
for i in range(10):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

You see that this is not exactly what we wanted. We're seeing numbers from 0 to 9, each one being printed on a new line. But what we wanted was all numbers from 1 to 10. Before we fix the code to produce the desired result, let's explain the bits and pieces of the code above.

What we just did was to use a so-called *for-loop*, probably the most common way to repeat things in Python. First we create an *iterator variable* `i` (we could have named any other variable name as well), which takes its value from the list of numbers specified by `range(10)`. If only one number `n` is provided to `range(n)`, it will enumerate all numbers from 0 to `n-1`. If instead two arguments are

provided, the first one determines the starting number, and the second one stands for the terminating number minus one—confusing, right?

So, in order to enumerate all numbers from 1 to 10, we have to write `range(1,11)`. Additionally, we can use the `end` keyword of the `print` function that allows us to print all numbers in one line, separated only by a single white space instead of each one on a new line.

```
for i in range(1,11):
    print(i, end=" ")
```

```
1 2 3 4 5 6 7 8 9 10
```

Voilà!

19.3. Just in case

Often we encounter a situation where we would execute some code **only if** certain conditions are met. In python, this is done with the `if` statement. For example, if we want to only print the even numbers in the range from 1 to 10, we could adapt the code from above as follows:

```
for i in range(1,11):
    if i % 2 == 0:
        print(i, "is even")
```

```
2 is even
4 is even
6 is even
8 is even
10 is even
```

You can read this as "if the remainder of dividing i by 2 is zero, then print 'i is even' ".

Now, we could also want to print a similar statement in the case that `i` is odd:

```
for i in range(1,11):
    if i % 2 == 0:
        print(i, "is even")
    else:
        print(i, "is odd")
```

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
10 is even
```

And, finally, we could also have more than just one condition. An if-statement allows for arbitrary many if-else clauses, with which we can formulate several different conditions by writing `elif` (shorthand for 'or else if'):

```
for i in range(1,11):
    if i % 2 == 0:
        print(i, "is even")
    elif i % 3 == 0:
        print(i, "is divisible by 3")
    else:
        print(i, "is odd")
```

```
1 is odd
2 is even
3 is divisible by 3
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is divisible by 3
10 is even
```

We now know when a number is even and when it is divisible by 3. But what about numbers that are *both* even *and* divisible by 3? We just add another condition to the `elif` statement and enclose each condition in parentheses, so that Python knows how things group together:

```
for i in range(1,11):
    if i % 2 == 0:
        print(i, "is even")
    elif (i % 2 == 0) and (i % 3 == 0):
        print(i, "is even and divisible by 3")
    else:
        print(i, "is odd")
```

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
10 is even
```

Why did this not work? The number 6 is even *and* divisible by 3! The reason is that the three statements (`if`, `elif`, and `else`) are being executed in the order that we wrote them down. That means, that Python will first check for each number whether it is even (and nothing more), and if it is,

it will follow the instruction to print it and go on to the next number. So, once we arrived at 6, the programm will only check if the number is even. That is not the desired result and we have to make a little change to it. We will switch the conditions in the `if` and `elif` statements:

```
for i in range(1,11):
    if (i % 2 == 0) and (i % 3 == 0):
        print(i, "is even and divisible by 3")
    elif i % 2 == 0:
        print(i, "is even")
    else:
        print(i, "is odd")
```

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even and divisible by 3
7 is odd
8 is even
9 is odd
10 is even
```

Now it works! Note that new never specified any conditions for the `else` statement. This is because whatever follows it will be executed in case none of the conditions in `if` or `elif` are met.

19.4. Functions

With more and more experience in programming, it is likely that your code will become more and more complex. That means that it will become harder to keep track of what every piece of it is supposed to do. A good strategy to deal with this is to aim for writing code that is *modular*: it can be broken down into smaller units (modules) that are easier to understand. Moreover, it is sometimes necessary to reuse the same code several times. It would, however, be inefficient to write the same lines over and over again. With your code being modular you can wrap the pieces that you need in several places into a *function*.

Let's look at an example! Assume, your (fairly) complex code involves calculating the sum of the squares of two numbers. In Python, we use the `+` operator to calculate sums and the `**` operator to raise a number to a certain power (`**2` for the square of a number).

```
x = 3
y = 5

sum_of_squares = x**2 + y**2
```

The variable `sum_of_squares` now contains the sum of squares of `x=3` and `y=5`. We can inspect the result by calling the variable:

```
sum_of_squares
```

```
34
```

Now, imagine that you would have to do the same calculation several times for different combinations of values for `x` and `y` (and always keeping in mind that this stands in for much more complex examples with several lines of code). We can code this in a function:

```
def func_sum_of_squares(x, y):
    return x**2 + y**2
```

Now, each time we want to calculate a sum of squares, we can do so by simply invoking

```
func_sum_of_squares(5,4)
```

```
41
```

And, of course, we could chose a shorter name for the function as well, although I would recommend to always use function names that make clear what the function does:

```
f = func_sum_of_squares

f(5,4)
```

```
41
```

19.5. Libraries you'll love

Luckily, you don't have to programm all functions by yourself. There is a huge community of Python programmers out there that works and collaborates on several *libraries*. A library is (more or less) simply a collection of certain functions (and some more, but we don't get into this here). This means, instead of writing a function yourself, you can rely on functions that someone else has programmed.

> **Caution**
>
> Whether a Python library or function does actually do what it promises is another story. Popular libraries with tens of thousands of users are very trust-worthy because you can be almost sure that someone would have noticed erroneous behavior. But it is certainly possible that badly-maintained libraries contain errors. So be prudent when using the code of others.

19.5.1. NumPy

One of the most popular Python libaries is NumPy for numerical computations. We will rely a lot on the functions in this library, especially in order to draw random samples—more on this later! To use the functions or variables from this library, they have to be *imported* so that you can use them. There are several ways to do this. For example, you can import the libary entirely:

```
import numpy
```

Now, you can use the (approximated) value of π stored in this library by typing

```
numpy.pi
```

```
3.141592653589793
```

A different way is to just import everything from the library by writing

```
from numpy import *
```

Here, the `*` stands for 'everything'. Now, to use the value of π we could simply type

```
pi
```

```
3.141592653589793
```

This is, however discouraged for the following reason: imagine we had another library, `numpy2` that also stores the value of π, but less precisely (e.g. only as `3.14`). If we write

```
from numpy import *
from numpy2 import *
```

We would have imported the variables holding the value of π from both libraries. But, because they have the same name `pi`. In this case, `pi` would equal `3.14` because we imported `numpy2` last. This is confusing and shouldn't be so! To avoid this, it is better to keep references to imported libraries explicit. In order not to have to type too much (we're all lazy, after all), we can define an alias for the library.

```
import numpy as np
np.pi
```

```
3.141592653589793
```

All functions of NumPy are now accessible with the prefix `np.`.

19.5.2. Pandas

[TODO]

19.5.3. Matplotlib

[TODO]

19.5.4. Summary

You can choose any alias when importing a library (it can even by longer than the library name) but certain conventions have emerged that you're encouraged to follow. Importing the most commonly used Python libraries for data-science tasks ("The data science triad"), use the following:

```
import numpy as np # for numerical computations
import pandas as pd # for tabular data
import matplotlib.pyplot as plt # for data visualization
```

We will use all three of them in the following chapters and you'll learn to love them.

Concepts covered

- variables
- types (integers, strings, lists, dictionaries)
- functions
- libraries, importing and aliases

Exercise for Week X

Finding Similarities and Differences in Folk Melodies with Python and Pandas

In this exercise we will have a look at the **Essen Folksong Collection (EFD)**. It is a database of Folksongs from all around the world gathered by the ethnomusicologist Helmut Schaffrath (1995). The collection can be downloaded from this website. It comes in the **kern format that is a table of note events where the rows correspond to event time.

We will answer a very specific question:

What can we learn about musical scales when we look at the notes in a musical piece?

For this purpose and also due to limited time, we need to make some simplifications. We will only consider **note counts**. That means, for this excercise we just count all the notes in a piece but do not care how long the notes are. Another way to say it is that we are just interested in the pitch dimension.

Notes in the EFD come as **spelled pitches**. Spelled pitches have three parts: 1. the diatonic step (C, D, E, F, G, A, or B) 2. possibly one or two accidentals (# or b) 3. the octave in which the note sounds

Moreover, we will reduce the pitches in a melody to **pitch classe**. This means we do not differentiate between the same pitches in different octaves, e.g. C4 and C5, nor will we distinguish between enharmonically equivalent pitches, such as F#3 and Gb3.

This way, each piece can be represented as a list of pitch classes that can then be counted.

Because it is not straight-forward to work with **kern scores in Python, the data was already transformed into DataFrame format and exported to `data.csv`.

First, we need to import sum libraries that contain the functions that we will use for the data analysis. If you do not have one or more of the packages installed, run `conda install packages-names` in a command-line tool (as adminstrator).

```
import numpy as np # for numerical computation (we need it to transpose melodies)
import pandas as pd # to organize our data in tabular format

import matplotlib.pyplot as plt # to visualize results
import seaborn as sns # more advanced visualization tools
# directly show the plots in the notebook
%matplotlib inline

import re # regular expressions: for pattern finding in strings
```

Step 1: Preprocessing

Read the data and create new column that contains a list of the pitches of the melody

```
data = pd.read_csv("../data/essen_data.csv", sep='\t', index_col=0)
data.head()
```

	region	title	key	DGIs	spelled_pitch
0	africa	Muwaschah Lamma Bada	g minor	['P4', 'M2', 'm2', 'M2', 'M-2', 'P1', 'm-2', '...	['D5', 'G5', 'A
1	mexico	CUCA 1	F major	['P1', 'P1', 'P4', 'M3', 'M-6', 'P1', 'P1', 'P...	['C4', 'C4', 'C
2	mexico	CUCA 2	F major	['P1', 'P1', 'P4', 'M3', 'M-6', 'P1', 'P1', 'P...	['C4', 'C4', 'C
3	mexico	CUCA 1	F major	['P1', 'P1', 'P4', 'M3', 'M-6', 'P1', 'P1', 'P...	['C4', 'C4', 'C
4	mexico	CUCA 2	F major	['P1', 'P1', 'P4', 'M3', 'M-6', 'P1', 'P1', 'P...	['C4', 'C4', 'C

Wee see that the (preprocessed) data comes with the features `region`, `title`, and `key`, and is represended as **directed generic intervals** (`DGIs`) and `spelled_pitches`.

For this excercise we will only work with the data in the `key` and `spelled_pitches` columns. The `spelled_pitches` entries look like a list of pitches but they are actually strings (`'['`, `','` and whitespaces are characters). Therefore we need to first transform it into a representation that we can work with. We will not go into details here but basically we just remove everything that we don't need from the string until only the spelled pitches are left.

```
data['spelled_pitches'] = \
    data['spelled_pitches'].str.replace("', '", " ").str.replace("['", "").str.replace("']", "
```

```
data.head()
```

	region	title	key	DGIs	spelled_pitch
0	africa	Muwaschah Lamma Bada	g minor	['P4', 'M2', 'm2', 'M2', 'M-2', 'P1', 'm-2', '...	D5 G5 A5 B-5
1	mexico	CUCA 1	F major	['P1', 'P1', 'P4', 'M3', 'M-6', 'P1', 'P1', 'P...	C4 C4 C4 F4
2	mexico	CUCA 2	F major	['P1', 'P1', 'P4', 'M3', 'M-6', 'P1', 'P1', 'P...	C4 C4 C4 F4
3	mexico	CUCA 1	F major	['P1', 'P1', 'P4', 'M3', 'M-6', 'P1', 'P1', 'P...	C4 C4 C4 F4
4	mexico	CUCA 2	F major	['P1', 'P1', 'P4', 'M3', 'M-6', 'P1', 'P1', 'P...	C4 C4 C4 F4

Let's also drop the columns that we don't need for this excercise.

```
del data['region']
del data['DGIs']
```

```
data.head()
```

	title	key	spelled_pitches
0	Muwaschah Lamma Bada	g minor	D5 G5 A5 B-5 C6 B-5 B-5 A5 A5 G5 G5 F#5 G5 A5 ...
1	CUCA 1	F major	C4 C4 C4 F4 A4 C4 C4 C4 F4 A4 F4 F4 E4 E4 D4 D...

	title	key	spelled_pitches
2	CUCA 2	F major	C4 C4 C4 F4 A4 C4 C4 C4 F4 A4 F4 F4 E4 E4 D4 D...
3	CUCA 1	F major	C4 C4 C4 F4 A4 C4 C4 C4 F4 A4 F4 F4 E4 E4 D4 D...
4	CUCA 2	F major	C4 C4 C4 F4 A4 C4 C4 C4 F4 A4 F4 F4 E4 E4 D4 D...

Let us inspect the data bit further with the `describe` method.

```
data.describe()
```

	title	key	spelled_pitches
count	9373	9372	9373
unique	7783	22	9015
top	Melodia instrumentalna	G major	B3 E4 F#4 G4 F#4 E4 B4 F#4 A4 G4 F#4 E4 D#4 E4...
freq	84	2831	4

It seems that we have some duplicates with different titles in the dataset. Normally, we should deal with this issue. For now we will treat them as separate songs. Also we can see that apparently there is missing key information for one song. Let's see where this happens.

```
data[ data['key'].isnull() ]
```

	title	key	spelled_pitches
1028	Yidui gezi xukongli fei	NaN	[]

Since we have only the title for this song, it doesn't make much sense to include it into our analysis. We will exclude it and drop this row from the DataFrame.

```
data.drop(1028, inplace=True)
```

```
data = data.reset_index(drop=True)
```

```
data.shape
```

```
(9372, 3)
```

Step 2: Extract the root and mode of the pieces and write them in new columns

Translate the keys into modes and pitch classes of roots. The easiest way might be to write a dictionary by hand that does the job.

```
data[['root', 'mode']] = data['key'].str.split(" ", expand=True)
```

```
data.head()
```

	title	key	spelled_pitches	root	mode
0	Muwaschah Lamma Bada	g minor	D5 G5 A5 B-5 C6 B-5 B-5 A5 A5 G5 G5 F#5 G5 A5 ...	g	minor
1	CUCA 1	F major	C4 C4 C4 F4 A4 C4 C4 C4 F4 A4 F4 F4 E4 E4 D4 D...	F	major
2	CUCA 2	F major	C4 C4 C4 F4 A4 C4 C4 C4 F4 A4 F4 F4 E4 E4 D4 D...	F	major
3	CUCA 1	F major	C4 C4 C4 F4 A4 C4 C4 C4 F4 A4 F4 F4 E4 E4 D4 D...	F	major
4	CUCA 2	F major	C4 C4 C4 F4 A4 C4 C4 C4 F4 A4 F4 F4 E4 E4 D4 D...	F	major

It is always a good idea to inspect the data to understand it better. What are the proportions of major and minor pieces in this dataset?

```
data['mode'].value_counts()
```

```
mode
major    8388
minor     984
Name: count, dtype: int64
```

```
data['mode'].value_counts() / len(data)
```

```
mode
major    0.895006
minor    0.104994
Name: count, dtype: float64
```

```
data['root'].value_counts()
```

```
root
G     2831
F     1878
C     1309
B-     664
D      648
A      553
g      370
E-     289
a      284
d      170
E      150
e      109
A-      56
f       19
c       17
b-       8
```

```
D-       6
B        4
d-       2
f#       2
b        2
e-       1
Name: count, dtype: int64
```

We don't want to distinguish between the roots of major and minor keys, so we just write all the roots as uppercase letters.

```
data['root'] = data['root'].str.upper()
```

```
data['root'].value_counts()
```

```
root
G     3201
F     1897
C     1326
A      837
D      818
B-     672
E-     290
E      259
A-      56
D-       8
B        6
F#       2
Name: count, dtype: int64
```

In order to transpose all the melodies to the same key, we need to know the pitch-class of each root. We will use a pragmatic approach and just explicitly state the information in a dictionary. It is common to define `'C'` as pitch class `0`.

```
roots_dict = {
    'G':7,
    'F':5,
    'C':0,
    'A':9,
    'D':2,
    'B-':10,
    'E-':3,
    'E':4,
    'A-':8,
    'D-':1,
    'B':11,
    'F#':6
    }
```

No we can translate the roots to pitch classes.

```
data['root'] = data['root'].map(roots_dict)
```

```
data.tail()
```

	title	key	spelled_pitches
9367	Die schoene Magdalena Was geschah an einem Mon...	G major	D4 D4 G4 G4 G4 E5 D5 B4 G4 D5 D5
9368	Das Maedchen und der Faehnrich 'Ach Tochter, l...	G major	D4 G4 A4 B4 C5 D5 B4 G4 G4 E5 E5
9369	Das Maedchen und der Faehnrich Es war ein reic...	G major	B4 D5 B4 D5 E5 D5 C5 B4 C5 D5 B4
9370	Der schwatzhafte Junggeselle Es waren drei Ges...	F major	C4 F4 E4 D4 C4 A4 G4 F4 C4 F4 E4
9371	Verschlafener Jaeger Es wollt ein Jaeger frueh...	A major	E4 A4 E4 E4 F#4 G#4 A4 F#4 F#4

Step 3: Create one new column for each pitch class in order to extract pitch-class counts

First, we transform the melody into a list of spelled pitches.

```
data['spelled_pitches'] = data['spelled_pitches'].str.split()
```

```
data.head()
```

	title	key	spelled_pitches	root	mode
0	Muwaschah Lamma Bada	g minor	[D5, G5, A5, B-5, C6, B-5, B-5, A5, A5, G5, G5...	7	minor
1	CUCA 1	F major	[C4, C4, C4, F4, A4, C4, C4, C4, F4, A4, F4, F...	5	major
2	CUCA 2	F major	[C4, C4, C4, F4, A4, C4, C4, C4, F4, A4, F4, F...	5	major
3	CUCA 1	F major	[C4, C4, C4, F4, A4, C4, C4, C4, F4, A4, F4, F...	5	major
4	CUCA 2	F major	[C4, C4, C4, F4, A4, C4, C4, C4, F4, A4, F4, F...	5	major

Next, we need a way to transform each pitch to a pitch class. To that end we define a function that takes a spelled pitch (a symbol such as `B-5`) and returns it's pitch class as a number between `0` and `12`.

```
def spelled_pitch_to_pitch_class(spelled_pitch):
    """
    This function transforms a spelled pitch, such as, `B-5`
    into a pitch class, a number between 0 and 12.

    A spelled pitch consists of three parts:
    1. Its diatonic step (C, D, E, F, G, A, or B)
    2. Potentially one or two accidentals (# or b)
    3. Its octave as a number.
    """

    # Remove octave by removing the last character in the string
    spelled_pitch_class = spelled_pitch[:-1]
```

```
    # Extract the diatonic step
    # First, we define a dictionary that associates
    # each diatonic step with a pitch class

    pitch_classes = {
        'C':0,
        'D':2,
        'E':4,
        'F':5,
        'G':7,
        'A':9,
        'B':11
    }

    # Extract accidentals
    # We define a regular expression that finds the three parts
    # of a spelled pitch class.
    match = re.match(r'(\w)(\#*)(-*)', spelled_pitch_class).group(1,2,3)

    # If we find a match, we get a tripel (step, sharps, flats)
    if match:
        step = pitch_classes[match[0]]
        sharps = len(match[1])
        flats = len(match[2])

    # The only thing left to do is to take the pitch class of the diatonic step,
    # add the number of sharps and subtract the number of flats
    # Finally, since pitch classes are always between 0 and 11, we take this number mod 12.
    return (step + sharps - flats) % 12
```

In the previous step we set up a function that converts spelled pitches into pitch classes. Now we can use it to count all the notes in a piece.

```
# First we set up an empty list that will later contain dictionaries of pitch-class counts for
countdicts = []

# Then we loop over all the rows (pieces) in our dataframe
for index, row in data.iterrows():

    # We replace the spelled pitches with pitch classes
    row['spelled_pitches'] = [spelled_pitch_to_pitch_class(pitch) for pitch in row['spelled_pi

    # Then we count the occurences of each pitch class in this list
    # We create an empty dictionary that will contain the pitch-class counts for the current p
    intcounts = {}

    # We iterate over all pitch classes and see if it is already in the `intcounts` dictionary
    for pitch_class in row['spelled_pitches']:
        # if not, set the count to 1
        if pitch_class not in intcounts.keys():
            intcounts[pitch_class] = 1
```

```
        # if yes, increment the count by 1
        else:
            intcounts[pitch_class] += 1
    # Finally, add the pitch-class counts dictionary to our list of pitch-class count dictiona
    countdicts.append(intcounts)
```

This is what the first 10 entries in the list of pitch-class count dictionaries looks like:

```
countdicts[:10]
```

```
[{2: 17, 7: 32, 9: 29, 10: 23, 0: 11, 6: 12, 3: 8, 5: 2, 4: 1},
 {0: 25, 5: 7, 9: 9, 4: 6, 2: 5, 7: 8, 10: 5},
 {0: 25, 5: 8, 9: 11, 4: 6, 2: 5, 7: 10, 10: 5},
 {0: 25, 5: 7, 9: 9, 4: 6, 2: 5, 7: 8, 10: 5},
 {0: 25, 5: 8, 9: 11, 4: 6, 2: 5, 7: 10, 10: 5},
 {0: 4, 5: 3, 9: 1, 2: 4, 7: 3, 4: 1},
 {5: 11, 0: 5, 7: 6, 9: 10, 10: 1},
 {2: 10, 11: 15, 9: 12, 7: 7, 6: 2, 0: 2},
 {7: 9, 11: 8, 9: 11, 2: 8, 0: 4, 6: 2, 4: 2},
 {7: 8, 9: 10, 11: 17, 0: 8, 2: 6, 4: 2}]
```

This is not really convenient. To handle it easier, we transform it to a DataFrame object and set all pitch classes to `0` if they do not occur in a piece.

```
counts = pd.DataFrame(countdicts).fillna(0)
```

```
counts.head(10)
```

	2	7	9	10	0	6	3	5	4	11	8	1
0	17.0	32.0	29.0	23.0	11.0	12.0	8.0	2.0	1.0	0.0	0.0	0.0
1	5.0	8.0	9.0	5.0	25.0	0.0	0.0	7.0	6.0	0.0	0.0	0.0
2	5.0	10.0	11.0	5.0	25.0	0.0	0.0	8.0	6.0	0.0	0.0	0.0
3	5.0	8.0	9.0	5.0	25.0	0.0	0.0	7.0	6.0	0.0	0.0	0.0
4	5.0	10.0	11.0	5.0	25.0	0.0	0.0	8.0	6.0	0.0	0.0	0.0
5	4.0	3.0	1.0	0.0	4.0	0.0	0.0	3.0	1.0	0.0	0.0	0.0
6	0.0	6.0	10.0	1.0	5.0	0.0	0.0	11.0	0.0	0.0	0.0	0.0
7	10.0	7.0	12.0	0.0	2.0	2.0	0.0	0.0	0.0	15.0	0.0	0.0
8	8.0	9.0	11.0	0.0	4.0	2.0	0.0	0.0	2.0	8.0	0.0	0.0
9	6.0	8.0	10.0	0.0	8.0	0.0	0.0	0.0	2.0	17.0	0.0	0.0

The DataFrame `counts` contains now the pitch-class counts for all pieces. We can see if the dimensions of `counts` and `data` fit.

```
counts.shape, data.shape
```

```
((9372, 12), (9372, 5))
```

But now, longer pieces weight more just because they contain more notes. To avoid that we have to normalize the DataFrame to get relative frequencies.

```
normalized = counts.div(counts.sum(axis=1), axis=0)
normalized.head(10)
```

	2	7	9	10	0	6	3	5	4	11
0	0.125926	0.237037	0.214815	0.170370	0.081481	0.088889	0.059259	0.014815	0.007407	0.000000
1	0.076923	0.123077	0.138462	0.076923	0.384615	0.000000	0.000000	0.107692	0.092308	0.000000
2	0.071429	0.142857	0.157143	0.071429	0.357143	0.000000	0.000000	0.114286	0.085714	0.000000
3	0.076923	0.123077	0.138462	0.076923	0.384615	0.000000	0.000000	0.107692	0.092308	0.000000
4	0.071429	0.142857	0.157143	0.071429	0.357143	0.000000	0.000000	0.114286	0.085714	0.000000
5	0.250000	0.187500	0.062500	0.000000	0.250000	0.000000	0.000000	0.187500	0.062500	0.000000
6	0.000000	0.181818	0.303030	0.030303	0.151515	0.000000	0.000000	0.333333	0.000000	0.000000
7	0.208333	0.145833	0.250000	0.000000	0.041667	0.041667	0.000000	0.000000	0.000000	0.312500
8	0.181818	0.204545	0.250000	0.000000	0.090909	0.045455	0.000000	0.000000	0.045455	0.181818
9	0.117647	0.156863	0.196078	0.000000	0.156863	0.000000	0.000000	0.000000	0.039216	0.333333

Now we are almoste done with transforming the data in order to answer our question. We still need to transpose all songs into the same key so that we can compare their pitch-class distributions. Let's think a moment about how this can be done. We have the pitch-class distribution of each song in `counts`, and we have `key`, `root`, and `mode` in `data`.

Let's say that we want to transpose all songs to the root C. C major and C minor pieces do not have to change. A piece in G major, for instance, has the root `7` and needs to be transposed to the root `0`. A piece in Bb minor has the root `10` and needs do be transposed to the root `0`. The easiest way to do this is to 'rotate' the pitch-class distributions by the negative amount of the root. Luckily, numpy provides the `roll` function to do exactly that. It takes an array (a vector) and rolls it by the specified amount. We do this for all songs in data and save the result in a new DataFrame `transposed`.

```
transposed = pd.DataFrame(
    [ np.roll( normalized.iloc[i,:], -data['root'][i] ) for i in range(len(data)) ]
    )

transposed.head(10)
```

	0	1	2	3	4	5	6	7	8	9	10
0	0.014815	0.007407	0.000000	0.000000	0.0	0.125926	0.237037	0.214815	0.170370	0.081481	0.08
1	0.000000	0.000000	0.107692	0.092308	0.0	0.000000	0.000000	0.076923	0.123077	0.138462	0.07
2	0.000000	0.000000	0.114286	0.085714	0.0	0.000000	0.000000	0.071429	0.142857	0.157143	0.07
3	0.000000	0.000000	0.107692	0.092308	0.0	0.000000	0.000000	0.076923	0.123077	0.138462	0.07
4	0.000000	0.000000	0.114286	0.085714	0.0	0.000000	0.000000	0.071429	0.142857	0.157143	0.07
5	0.000000	0.000000	0.187500	0.062500	0.0	0.000000	0.000000	0.250000	0.187500	0.062500	0.00
6	0.000000	0.000000	0.333333	0.000000	0.0	0.000000	0.000000	0.000000	0.181818	0.303030	0.03
7	0.000000	0.000000	0.312500	0.000000	0.0	0.208333	0.145833	0.250000	0.000000	0.041667	0.04
8	0.000000	0.045455	0.181818	0.000000	0.0	0.181818	0.204545	0.250000	0.000000	0.090909	0.04
9	0.000000	0.039216	0.333333	0.000000	0.0	0.117647	0.156863	0.196078	0.000000	0.156863	0.00

We can already observe that none of the first 10 songs in the dataset has the tritone (pitch class 6), and only one has a minor third (pitch class 3) or a minor seventh (pitch class 10).

It would be nice not having to work with to DataFrames, `data` and `transposed`, so we combine (concatenate) them in a new one, just called `df`.

```
df = pd.concat([data, transposed], axis=1)
```

```
df.shape
```

```
(9372, 17)
```

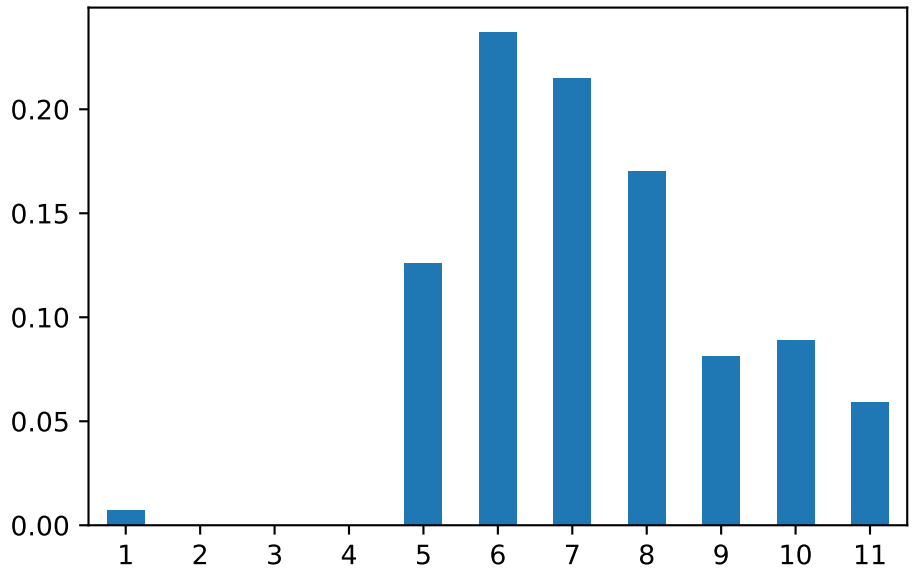
```
df.head()
```

	title	key	spelled_pitches	root	mode	0
0	Muwaschah Lamma Bada	g minor	[D5, G5, A5, B-5, C6, B-5, B-5, A5, A5, G5, G5...	7	minor	0.0
1	CUCA 1	F major	[C4, C4, C4, F4, A4, C4, C4, C4, F4, A4, F4, F...	5	major	0.0
2	CUCA 2	F major	[C4, C4, C4, F4, A4, C4, C4, C4, F4, A4, F4, F...	5	major	0.0
3	CUCA 1	F major	[C4, C4, C4, F4, A4, C4, C4, C4, F4, A4, F4, F...	5	major	0.0
4	CUCA 2	F major	[C4, C4, C4, F4, A4, C4, C4, C4, F4, A4, F4, F...	5	major	0.0

Step 4: Plot your first pitch class histogram and pitch class distribution

1. Choose an example piece.
2. What do you expect to see?
3. Plot a pitch class histogram in chromatic order.

```
piece = df.iloc[0,-11:]
piece.plot.bar(rot=0);
```



Step 5: Plot the averaged pitch class distribution for the major and the minor mode

2. Plot the averaged distributions.
3. Can you show everything in one figure?
4. Would it also make sense to plot averaged pitch class histograms?

```
melted = df.melt(id_vars='mode',
                 value_vars=[0,1,2,3,4,5,6,7,8,9,10,11],
                 var_name='pitch_classes',
                 value_name='relative_frequencies'
                )
```

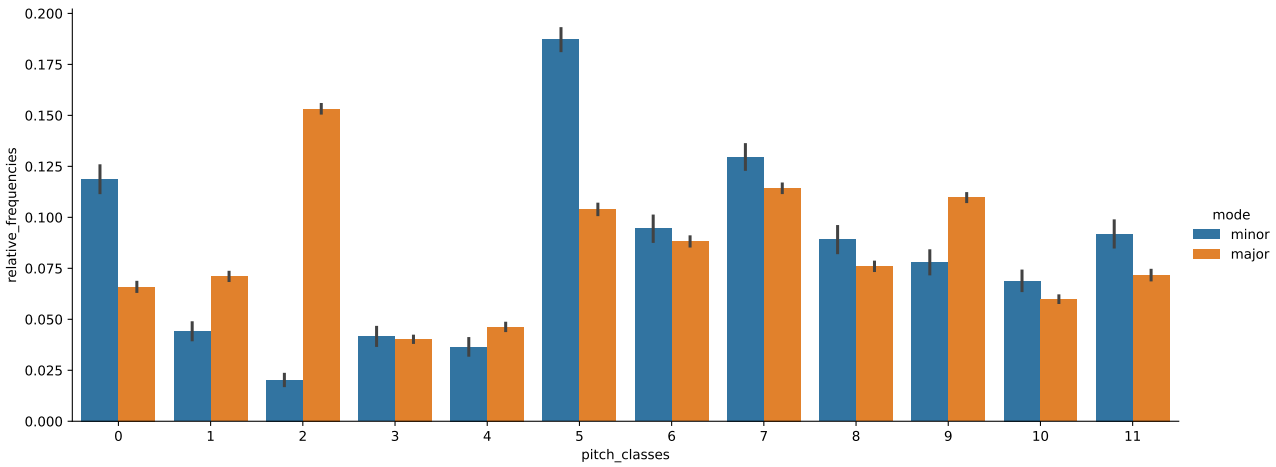
```
melted.head()
```

	mode	pitch_classes	relative_frequencies
0	minor	0	0.014815
1	major	0	0.000000
2	major	0	0.000000
3	major	0	0.000000
4	major	0	0.000000

```
melted.shape
```

```
(112464, 3)
```

```
sns.catplot(data=melted,
            x='pitch_classes',
            y='relative_frequencies',
            hue='mode',
            kind='bar',
            aspect=2.5
           );

plt.show()
```

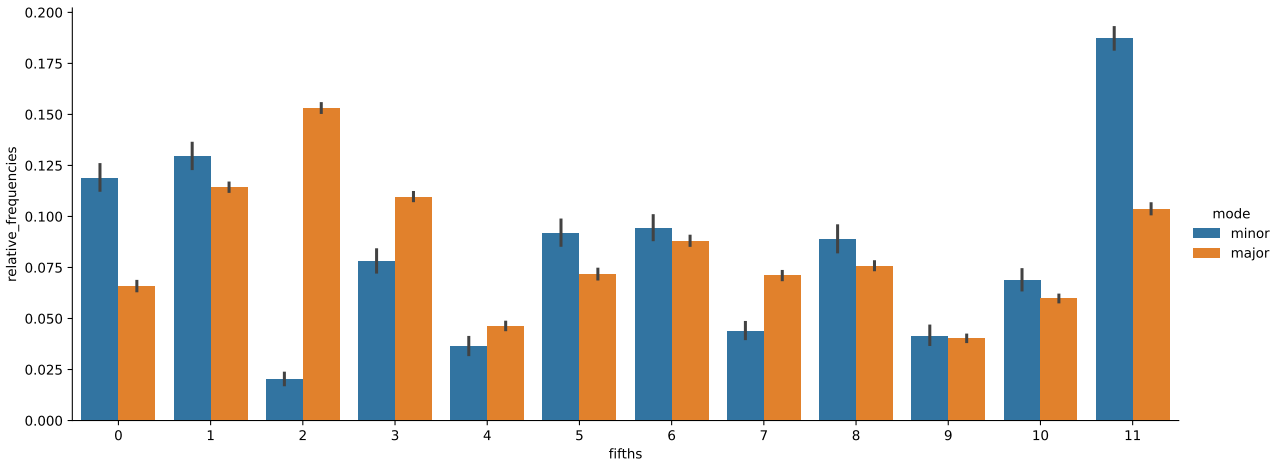


Step 6: Plot the averaged distributions in fifths ordering

1. Create the plot.
2. What do you see?

```
melted['fifths'] = melted['pitch_classes'] * 7 % 12
```

```
sns.catplot(data=melted,
                x='fifths',
                y='relative_frequencies',
                hue='mode',
                kind='bar',
                aspect=2.5
               );
```

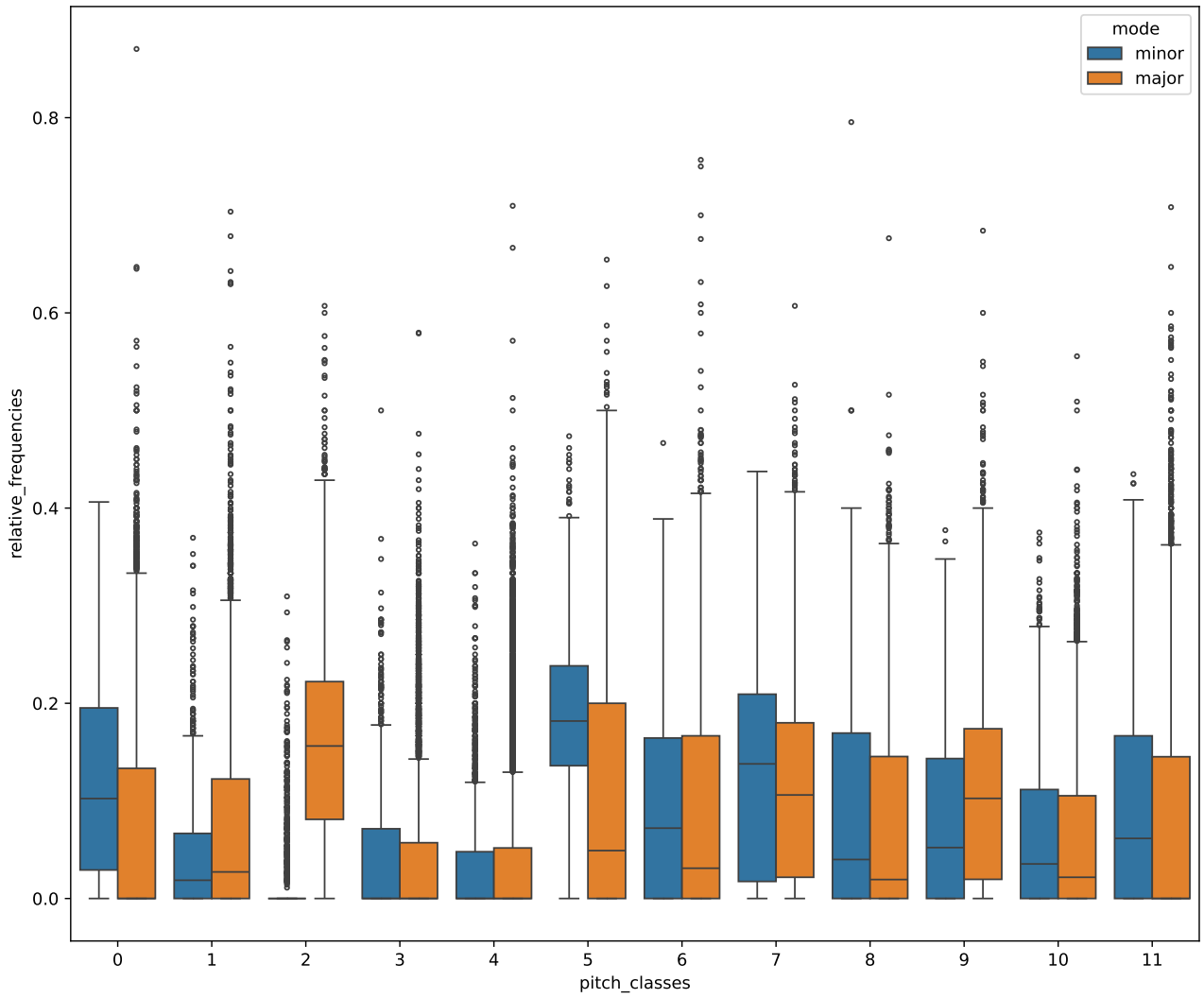


Step 7: Extend the plot above to show the diffusion of each pitch class

1. Decide to either use error bars, boxplots, or violin plots. What is the difference between them? Violin plots are of cause the most fancy figures…
2. Describe what you see.

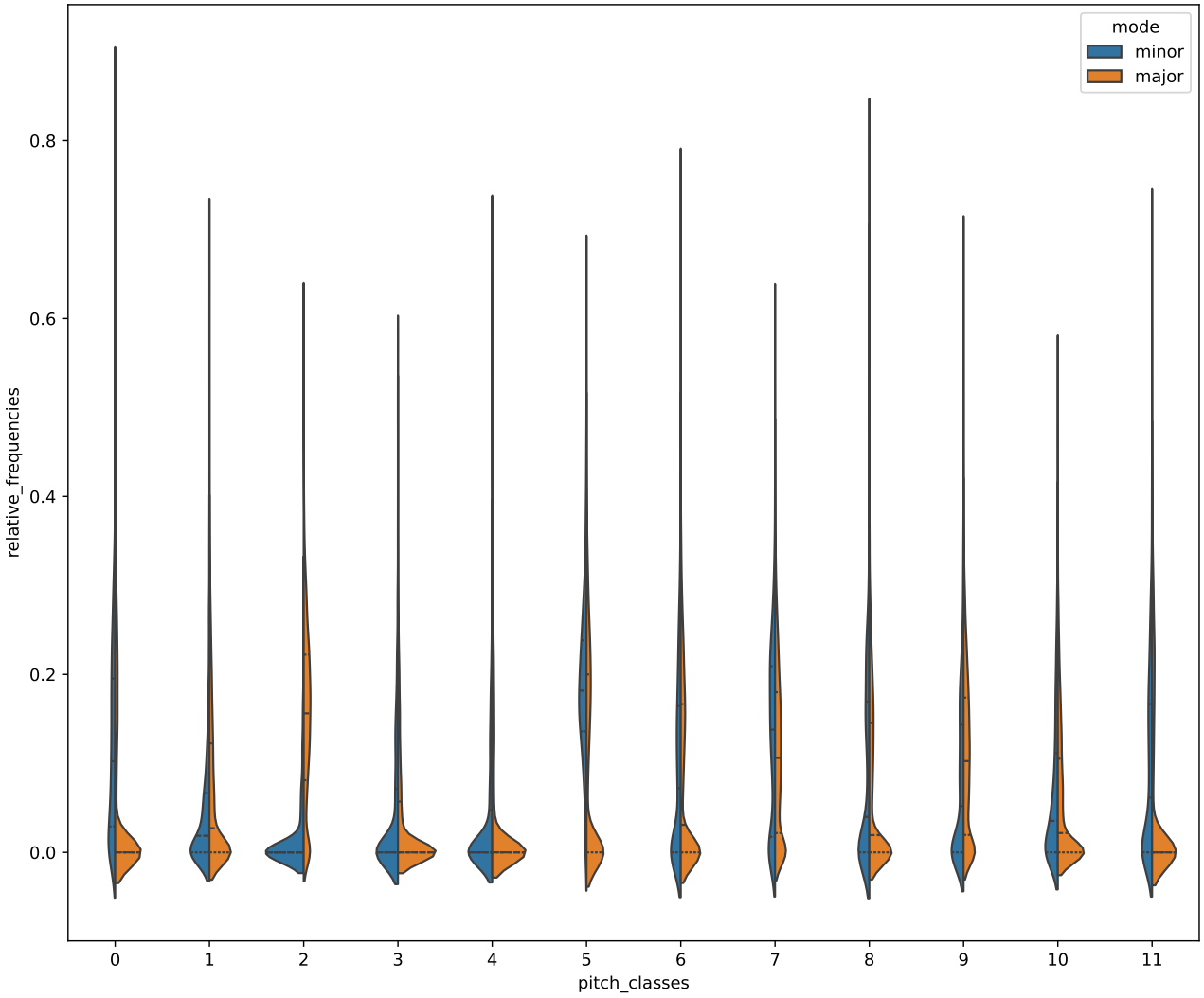
```
plt.figure(figsize=(12,10))
sns.boxplot(
    data=melted,
    x='pitch_classes',
    y='relative_frequencies',
    hue='mode',
    fliersize=2.5
);
```

Step 7: Extend the plot above to show the diffusion of each pitch class



These boxplots show already much more! For example, they reveal that there are many outliers which we can't see in the bar plot. https://www.autodeskresearch.com/publications/samestats

```
plt.figure(figsize=(12,10))
sns.violinplot(
    data=melted,
    x='pitch_classes',
    y='relative_frequencies',
    hue='mode',
    inner='quart',
    split=True
);
```


mode
minor
major
relative_frequencies
0.0
0.2
0.4
0.6
0.8
0
1
2
3
4
5
6
7
8
9
10
11
pitch_classes

REFERENCES

Budge, Helen. 1943. "A Study of Chord Frequencies Based on Music of Representative Composers of the Eighteenth and Nineteenth Centuries." PhD thesis, Columbia University.

Burgoyne, John Ashley, Ichiro Fujinaga, and J. Stephen Downie. 2015. "Music Information Retrieval." In *A New Companion to Digital Humanities*, 213–28. John Wiley & Sons, Ltd. https://doi.org/10.1002/9781118680605.ch15.

Eerola, T. 2025. *Music and Science: A Guide to Empirical Music Research*. SEMPRE Studies in the Psychology of Music. London, UK: Routledge.

Inskip, C., and F. Wiering. 2015. "In Their Own Words: Using Text Analysis to Identify Musicologists' Attitudes Towards Technology." In *Proceedings of the 16th International Society for Music Information Retrieval Conference, Malaga, Spain. (2015)*. Malaga, Spain: 16th International Society for Music Information Retrieval Conference.

Jeppesen, Knud. 1927. *The Style of Palestrina and the Dissonance*. 1st ed. New York: Oxford University Press.

Müller, Meinard. 2015. *Fundamentals of Music Processing: Audio, Analysis, Algorithms, Applications*. Springer International Publishing. https://doi.org/10.1007/978-3-319-21945-5.

Norman, Philip B. 1945. *A Quantitative Study of Harmonic Similarities in Certain Specified Works of Bach, Beethoven, and Wagner*. C. Fischer, Incorporated.

Schaffer, Kris. 2016. "What Is Computational Musicology?" https://medium.com/@krisshaffer/what-is-computational-musicology-f25ee0a65102.

Sethares, William A. 2005. *Tuning, Timbre, Spectrum, Scale*. 2nd ed. London: Springer.

Shanahan, Daniel, John Ashley Burgoyne, and Ian Quinn, eds. 2022. *Oxford Handbook of Music and Corpus Studies*. Oxford: Oxford University Press.

Wing, Jeannette M. 2006. "Computational Thinking." *Commun. ACM* 49 (3): 33–35. https://doi.org/10.1145/1118178.1118215.