Chapter 1

Case study: Zipf's Law and Music

Max Ward (20748588)

1.1 Music as Language

Zipf's law was introduced in Chapter 5. To recapitulate, it posits that the frequency of a word is inversely proportional to its rank. The rank of a word being defined as its index in a frequency table. The canonical example of this is the words of long piece of literature, usually a novel. It is a classical example of a scale-free distribution, and is defined by $f = r^{-a}$ where f is frequency and r is rank. A frequency distribution is said to be Zipfian if $a \approx 1$. Because of this, when plotted on a log-log scale (equivalent to taking the log of both sides), a Zipfian distribution appears to be a line with slope $m \approx -1$. Though Zipf's law is typically described in terms of novels, and thus written language, it actually refers to any natural language. Natural language is generally regarded to be any non-contrived language—which is to say that its use and invention is unpremeditated. Clearly most written and spoken languages are natural languages, however, written and performed music is also a type of natural language.

Figure of a log log zipf graph here

Music, among other art forms, is often regarded as communication. Like other forms of human communication it has structure and rules. While I shall not give a formal definition of these, the reader may refer to `http://en.wikipedia.org/wiki/Music_theory` for a concise overview. This chapter investigates Zipf's law in music. This is done by using the MIDI file type, which is a relatively simple and widely used way to encode digital music.

Exercise 1.1. *There are specific parts of the human brain involved in language. For example, damage to Broca's area can profoundly affect ones ability to understand spoken and written language. Does an analogous region exist for processing music? (Hint: try searching for 'Amusia'.)*

1.2 The MIDI Format

The MIDI file format comprises a header and a series of one or more tracks. Ever MIDI track also contains a header, which is followed by a sequence of events. There are many event types which refer to any time related possibility. However, the only two which we

consider in this investigation are the 'Note On' and 'Note Off' events. A Note On event has a time, a velocity, and a note. The time is when it is played, the velocity is the force with which it is played, and the note is the specific sound frequency of the event. It should be noted that velocity is not volume. It is possible than a MIDI instrument could have a linear velocity-volume relationship, but it could be exponential, logarithmic, or anything else. There are 128 distinct notes in the MIDI format, these are analogous to an extended piano (which has only 88 keys), with each increment indicating a semitone increase in sound frequency.

Figure of a piano keyboard here.

Parsing the MIDI file format can be difficult, as such, it is often relegated to a library. Python does not come with an easy to install MIDI parsing library. However, an open source effort called `python-midi` is available on Github under the MIT licence. It can be found here `https://github.com/vishnubob/python-midi`. Download or clone the repository and use the source files in your project. Once they are in the same folder, you should be able to import them like any Python library. I have provided a basic example that loads a MIDI file, given a file name from the command line. The `fileio` module is part of `python-midi`. It contains functions related to reading and parsing MIDI files.

```
import sys
import fileio


def main(name, fname, *args):
    pattern = fileio.read_midifile(fname)
    print pattern


if __name__ == '__main__':
    main(*sys.argv)
```

The way MIDI patterns are represented in `python-midi` is straightforward. A pattern is a list of tracks. A track is a list of events. Every event has a type, and associated metadata. All of these classes also have the `__repr__` method overloaded. This makes it easy to determine what information is available. A quirk of some older MIDI files is that a Note On event with velocity 0 is equivalent to the corresponding Note Off event. Keep this in mind when experimenting with MIDI files.

The Classical Archive has some excellent MIDI encodings of classical music (available at `http://www.classicalarchives.com/midi.html`). You may also wish to find some songs which you enjoy, many people have made their MIDI versions of songs freely available on the Internet. There is also software that allows the user to write MIDI music using a graphic environment. If you are so inclined, it is possible to analyse your own compositions.

Exercise 1.2. *Research the MIDI format. How hard do you think it would be to write your own parser? Remember, if you improve* `python-midi`*, submit a pull request to Github!*

Exercise 1.3. *Attempt to write a script that outputs the number of distinct notes in a MIDI file. My solution is available at URL.*

1.3 Words and Music

Zipf's law refers to the frequency of words found in a large corpus of natural language. It is hard to see how the concept of a word could apply to musical compositions. Of course, any interpretation of a word in music is necessarily arbitrary and somewhat open to interpretation. Despite this, I shall now describe several feasible candidates. Clearly the most obviously irreducible component of a musical composition is the note. Building upon this starting point, one might analyse the velocities, lengths, or the spacing between similar notes. More complex interpretations may attempt to group notes into structures like chords or melodies. I shall, however, start with a much simpler interpretation.

1.3.1 Notes

As I have said, we can break a piece of music up into notes. If we define a word as a single note, we can then collect the frequencies of these notes, and thence subject them to Zipfian analysis. This necessitates going through every MIDI event in every MIDI track for any given pattern. The `python-midi` package makes this quite easy, as tracks and events are represented as objects that inherit from `list`. Here is a simple function that returns all the notes (in no particular order) given the path to a MIDI file.

```
def get_notes(fname):
    pattern = fileio.read_midifile(fname)
    notes = []
    for track in pattern:
        for event in track:
            if event.name == "Note On" and event.data[1] > 0:
                notes.append(event.data[0])
    return notes
```

The first line reads and parses the MIDI into a pattern. The follow two loops examine every event in the pattern. Every time a genuine Note On event is spotted (remember that a Note On event with velocity 0 is actually a Note Off event) it is added to the `notes` list. It should be pointed out that event data is stored in the `event.data` member. For Note On events, index zero is the MIDI note, and index one is the velocity. After executing this function, we now have everything we need to calculate frequency as a function of rank.

1.3.2 Consecutive Notes

A more versatile way of using note frequencies is to consider sequences of consecutive notes. This is equivalent to sorting all the notes in the MIDI file by order of appearance, then running a sliding window of fixed size over all the notes. This will result in a much larger corpus of possible words. For example, a window size of two will have 128^2 possible words, every combination of two MIDI notes. Having a window size of one is the same as the approach previously described in Section 1.3.1. Having a too large window size can be a problem, however, as it often results in having many words with very low frequency. I found that a window size of two was typically the best. Very long pieces of music may work with larger window sizes, however.

Figure of sliding window

Exercise 1.4. *See if you implement the sliding window note analysis technique described in this section. Note On objects have a member called* `tick`*, which is the number of ticks before they appear in the MIDI song. You may wish to use this for sorting. My solution can be found at URL.*

1.3.3 Note and Velocity Pairs

I have outlined how we could use the occurrence of notes as an interpretation of words for Zipfian analysis. However, a Note On event has two components; the note and its velocity. Another reasonable interpretation of a word is to use a tuple of these data. A quirk of the `python-midi` package is that `event.data` is actually a list. Because of this, I have used the built-in function `tuple` to convert it to a tuple in the code snipped presented below.

```
def get_notevelocities(fname):
    pattern = fileio.read_midifile(fname)
    notes = []
    for track in pattern:
        for event in track:
            if event.name == "Note On" and event.data[1] > 0:
                notes.append(tuple(event.data))
    return notes
```

Exercise 1.5. *Can you think of any problems with this method? Would using a sliding window over note and velocity pairs be a good idea?*

1.3.4 Note Intervals

Thus far, all the word interpretations I have described deal with discrete events directly. As an alternative, we may wish to take a more high level approach. For every MIDI note frequency that appears in a MIDI pattern, the intervals between specific notes could be recorded. For example, if the note code 60 appears at tick 110, then later at 120, then again at 135, we would record intervals of 10 and 15. Such intervals could be collected for every note in the MIDI pattern, and collated into one list. We can then calculate the rank and frequency of note intervals for further analysis.

Exercise 1.6. *Implement the* `get_noteintervals` *function. Modifying the* `get_notes` *function might make this easier. Don't forget to sort all MIDI events by tick first. (Hint: use a dictionary to store the last occurrence of every note code.)*

Exercise 1.7. *Can you think of any other aspects of music that might reasonably conform to Zipf's law? If you can think of any, implement a getter function like those present in this section.*

Going to explain all the different 'words'
exercises will be the implement as many types as possible
remind to sort by ticks

1.4 Scale-free Music

This is the results section
show some log log plots

show some correlations

ask the reader to implement their own stuff

Our implementation of Sugarscape aims to study the effect of taxation on the wealth of a society. We want to show how extreme under- or over-taxation can affect the society and its individual agents, and what happens in between these two extremes. The model tests a "flat tax" system where every agent gets taxed a constant rate (say 10% of its total wealth) and the tax pool is redistributed evenly among all the agents. We recreate the original Sugarscape and expand on it with the end goal of determining whether it is possible to shrink the wealth gap without crippling the society.

1.5 The Beauty of Context

Talk about how Zipf's law may be a precondition for beautiful music

reference that paper that used it as a fitness function

Talk about how pink noise is nicer than brown/white Ask the reader to listen to all the kinds of noise

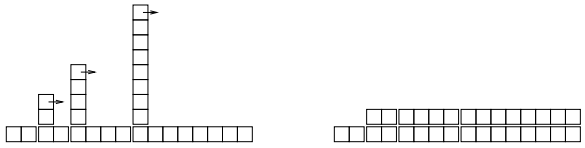

Figure 1.1: Histogram of wealth with no tax.

To compare the effect of taxation on wealth distribution, we need a metric that measures how distributed or flat a certain wealth distribution is. We use the Gini coefficient, which is often used in economics to measure the wealth gap (see `http://en.wikipedia.org/wiki/Gini_coefficient`). The Gini coefficient is between 0 and 1, with 0 the measurement of a perfectly uniform distribution, and 1 the measurement of a distribution with complete inequality.

Figure 1.1 shows a histogram describing the wealth distribution when there is no tax system in place. For most initial conditions without taxation, the Sugarscape quickly develops a long-tailed distribution of wealth, skewed to the right. In these cases, some agents die quickly, particularly in an environment with many agents or one with low sugar regrowth rate. The separation between the rich and the poor is significant, and there aren't many agents occupying the middle ground. This is seen in real life in societies where there is no tax structure and there isn't much of a middle class.

1.6 Reductionism or Holism?

Written English and Music both share Zipfian properties. A holist would say that Zipf's law is an underlying property of natural languages. What would a reductionist say? Remember that we have reduce language and music alike down too words. This is much the same way a physicist might modle suns and planets like point masses. Can we use this property to make predictions about other natural languages. What would an instrumentalist say? How about a realist?

The construction hypothesis; language and music have these properties because they are constructed.

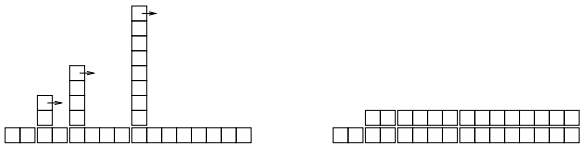

Figure 1.2: Histogram of wealth, with tax.

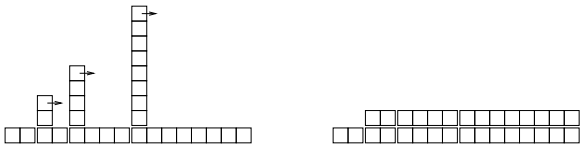

Figure 1.3: The Gini coefficient versus the tax rate.

Figure 1.2 shows the effect of a relatively high tax rate. The agents have a similar amount of sugar, and the economy has a low Gini coefficient, 0.02.

Figure 1.3 shows that higher taxes in general result in lower Gini coefficients. This makes sense, since the point of our tax system is to redistribute wealth.

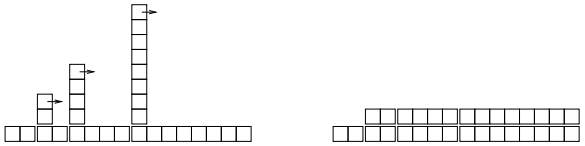

Figure 1.4: Mean wealth versus tax rate.

In this model, perfect equality comes at a price. With no taxation the mean wealth was 358; with a 20% tax rate it drops to 157. Figure 1.4 shows the effect of tax rate on wealth; mean wealth gets smaller as taxes get higher.

1.7 Instrumentalism and Music

It's up to a society to determine its ideal wealth distribution. In our model, there is a conflict between the goals of maximizing total wealth and minimizing inequality.

One way to reconcile this conflict is to maximize the wealth of the bottom quartile. Figure 1.5 shows the mean wealth of the poorest 25% for a range of tax rates. The optimal tax rate is around 4%. At lower rates, there is more total wealth, but the poor do not share it. At higher rates, the poor have a bigger share of a smaller pie.

Of course, this result depends on the details of our Sugarscape, especially the model of productivity. But this simple model provides a way to explore relationships between wealth creation, taxation and inequality.

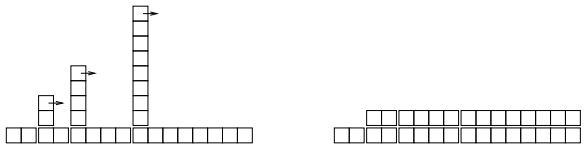

Figure 1.5: Bottom quartile value versus tax rate. At 4% the average wealth of the bottom quartile is maximized.

Exercise 1.8. *You can download our implementation of Sugarscape from* `thinkcomplex.com/Sugarscape.zip`*. Launch it by running* `Gui.py`*. The sliders allow you to control the parameters of the simulation. Experiment with these parameters to see what effect they have on the distribution of wealth.*

Exercise 1.9. *You can download our implementation of Sugarscape from* `thinkcomplex.com/Sugarscape.zip`*. Launch it by running* `Gui.py`*. The sliders allow you to control the parameters of the simulation. Experiment with these parameters to see what effect they have on the distribution of wealth.*