American University - DATA 312

Instructions for Homework 3-B

Unit 3 – Music analysis
Part B – MIDI files

Pre-requisite work:
Please complete Worksheet 3-A before starting this assignment.

Make sure that the following R libraries are installed using install.packages("<library>",dependencies=TRUE):

1. tidyverse
2. tabr
3. **NEW!** tuneR (watch the capitalization!)

Download the MIDI files from Canvas that are included with this assignment. There are four (4) of them!

1. `bach_846_format0.mid`: Bach Prelude and Fugue in C major BWV 846: Source http://www.piano-midi.de/
2. `justin_rubin_lyric.mid`: A duodecaphonic composition by Justin Rubin: Source: https://www.d.umn.edu/~jrubin1/JHR%20Dodec%202.htm
3. `improv.mid`: An improvisation on single piano by Edwin Robinson, recorded on a Casio Privia keyboard, passed through the LMMS composition program, and then converted to MIDI using the LMMS-to-MIDI online converter: https://www.lynxwave.com/LMMStoMIDI/LMMStoMIDIConverter.html
4. `improv2.mid`: An improvisation on single piano by Edwin Robinson, recorded on a Casio Privia keyboard (directly)

Before you start in on the analysis, have your computer play each of these MIDI files. That will help you understand some the musical features that we'll be exploring.

Objectives:

The MIDI file format is commonly used by musicians to control electronic instruments. Although perhaps you've only heard about MIDI as being bad video game music, because it originated in the early 1980s, it's the main format for exchanging musical data. Perhaps more surprisingly, most instruments still use the original format specification, even though it's nearly forty years old! That's because what MIDI represents is pretty basic: which instruments are being used, the timing of the notes being played, their pitches, and a record of various settings for the instruments. The settings on the instruments are actually optional and are specific to each manufacturer, so if you want to use a different instrument it's easy to switch. (This is why MIDI files usually sound cheesy when played from your computer. It doesn't know how to interpret all those settings, so it just ignores them and plays with a basic set of instruments.)

Although MIDI files were first used mostly on electronic instruments, they're now used by many music composition computer programs, too. While MIDI is not sheet music, and sheet music is not MIDI, the two are closely related. Some electronic composers just use MIDI, since that's a format that electronic musicians are comfortable using! MIDI can represent timing subtleties that sheet music cannot show (without becoming unreadable), and sheet music can instruct the musician about intended effects that MIDI cannot. Regardless, there are many different sources of MIDI files, which is why we'll be looking at four of them.

MIDI files are composed of a sequence of "events", which are pre-defined commands for the instruments or carry information about the piece of music. As we'll see in this worksheet, if you're just interested in the sheet music that goes with a MIDI file, you can ignore most of the messages and just focus on the "Note On" events that tell you what notes to play. Different instruments and computer programs that produce MIDI files all use these "Note On" events in the same way, but their usage other events can differ substantially. That's why this worksheet has uses different MIDI files, mostly to show you how these other events work.

The tabr library that we used in HW3A can read MIDI files. It turns MIDI into a tibble data frame, in which each row is an event, and the columns delineate different fields in the events. Not all fields are used by all events, and some fields are optional. This makes for a pretty convenient place to start our analysis, so lets get to it!

Work Process:

1. As usual, start by clearing the workspace and loading libraries:

```
rm(list=ls())
gc()
library(tidyverse)
library(tabr)
library(tuneR)
```

2. The next step is to read the MIDI files into your R session. Make sure to adjust the paths to the files so that R finds them. I'm assuming that RStudio is in the same folder as the files, which probably is not the case for you. (Remember that RStudio can often help you find a file: type the quotes, and then hit TAB between them. This should bring up a folder menu to help you locate the files. If not, just ask! I'll help you figure it out.)

```
bach<-read_midi('bach_846_format0.mid')
edwin_improv<-read_midi('improv.mid')
edwin_improv2<-read_midi('improv2.mid')
duodecatonic<-read_midi('justin_rubin_lyric.mid')
```

Just as a note: none of these files are particularly big, so carrying them all in your R session won't cause any problems. Furthermore, it makes comparing them rather easy!

Once you've done this, have a look at each of the data frames you've created!

3. MIDI files contain quite a bit of interesting data within them besides the music. Each row in the MIDI data frame is called an "event". You can get a quick summary of what the different kind of events are by something like

```
bach %>% count(event)
```

One thing to note is that the "Note On" and "Note Off" events are paired -- there are usually exactly the same number of each of these. Except when there's not.... this can happen if there are a few notes that are held past the end of the piece. (I guess the instruments in that case have to figure out what to do....)

4. What would be nice is to have a single table that compares the event counts you made in Step 3 across all four files. Since the event names are determined by the MIDI standard, they're the same across all files. This means you can join the count tables. Right off, there's a problem because the count function makes the `n` column by default. You can't tell the different files apart, and the join command will get confused. But you can fix this by telling count to name your new column something else:

```
bach %>% count(event,name='bach')
```

or

```
count(bach,event,name='bach')
```

I suggest a full_join to do your joining, because if some file doesn't have a given event, we want that entry filled with NA. You can start off with two of these:

```
count(bach,event,name='bach') %>%
  full_join(count(edwin_improv,event,name='edwin_improv'))
```

and then continue for the rest:

```
count(bach,event,name='bach') %>%
  full_join(count(edwin_improv,event,name='edwin_improv'))%>%
  full_join(count(edwin_improv2,event,name='edwin_improv2'))%>%
  full_join(count(duodecatonic,event,name='duodecatonic'))
```

That should give you a nice table to investigate. There are a few things that ought to jump out at you. For one, the Bach piece has a much bigger header. The first few events are all metadata about the file, its copyright info, as well as setup for the instrument. You can also see that the two files produced by computer composition programs (the bach and duodecaphonic files) differ substantially from the recorded ones (edwin_improv and edwin_improv2). Quite a bit of this is timing information added by the composer. This information actually is present in the recorded files, but is implicit in the "Note On" and "Note Off" events.

5. In order to dig deeper into the files, it's helpful to make a bunch of comparison tables like what we did in Step 4. However, what we did in Step 4 is rather single-use. The tidy data perspective is that we should reorganize our data a bit so that it makes it much easier to do many different kinds of tasks. So, let's redo Step 4 using a tidy approach...

The way to do this is to add another column to our data that stores the file that the events (rows) came from. For instance, we can add a `name` column like so

```
mutate(bach,name='bach')
```

Then stack all the rows from all the files into one big table:

```
songs<-bind_rows(mutate(bach,name='bach'),
      mutate(edwin_improv,name='edwin_improv'),
      mutate(edwin_improv2,name='edwin_improv2'),
      mutate(duodecatonic,name='duodecatonic'))
```

Recalling that count pays attention to groups within a table, we essentially want to group_by the name column and then count. This does the job:

```
songs %>% group_by(name) %>% count(event)
```

but it's a bit unreadable. To make the same kind of table we made in Step 4, we need to make each file its own column. This is called "pivoting": we want to pivot the table so that it becomes a wider table, and we want to use the `name` column to build out new columns:

```
songs %>% group_by(name) %>%
 count(event) %>%
 pivot_wider(names_from=name,values_from=n)
```

6. I think you'll agree that the tidyverse way (Step 5) took more effort than the simpler way (Step 4) for analyzing the events across files. But the benefit of the tidyverse way is that we can reuse the songs table to do other analysis easily. For instance, let's compare how "MIDI channels" are used across the files:

```
songs %>% group_by(name) %>%
 count(channel) %>%   # Notice that this is the only line we had to change!
 pivot_wider(names_from=name,values_from=n)
```

The Bach MIDI has several channels, while the others are single channel files. That is not to say that only one note plays at a time, just that there's only one instrument. The Bach piece is largely one instrument, but it's an organ... so... well...

You can do this same analysis for "track", "type", "duration", or any other column from the data. Try to discover some interesting differences between the files!

7. Just to show off the power of R for doing analysis of the data, let's figure out what the parameter1 and parameter2 columns mean. When I first had the files loaded, these were a bit of a mystery. The R tools were helpful in figuring out what they are! Let's plot them:

```
songs %>% group_by(name) %>% ggplot(aes(x=name,y=parameter1)) + geom_point()
```

```
songs %>% group_by(name) %>% ggplot(aes(x=name,y=parameter2)) + geom_point()
```

What a mess! You can try some other geoms, as they might look a little better. Although not everyone likes "violin" plots, they're not a half bad way to get a quick look at distributions of values, and well, we are doing analysis of music...

```
songs %>% group_by(name) %>% ggplot(aes(x=name,y=parameter1)) + geom_violin()
```

8. If you look at the songs table itself, you might get the impression that parameter1 has something to do with note pitch. A quick plot should help answer that by plotting pitch against parameter1:

```
songs %>% group_by(name) %>% ggplot(aes(x=pitch,y=parameter1)) + geom_point()
```

You should see that there's clearly a relationship, but... there are apparently several lines, maybe? That's actually true: there are several octaves represented.

9. We can convince tabr to do note frequency lookups for each pitches, so that we can compare the values of parameter1 to the note frequencies in Hz. However, tabr is super picky about what you feed it for pitches: it doesn't like NAs or the like. So let's remove those from the analysis with a filter. You can use the is.na function to detect NAs. If you want anything that's not NA, you can use the "!" character to mean NOT. Therefore,

```
songs %>% filter(!is.na(pitch))
```

is just the MIDI events that have pitches with them. To do a note frequency lookup, recall that in HW3A, the as_music_df function made a data table with the `freq` column listing note frequencies. Thus

```
songs %>% filter(!is.na(pitch)) %>%
  mutate(as_music_df(pitch))
```

will give you a table with all the columns from before, along with various new ones... including the `freq` column we wanted. It takes a few seconds on my computer to run, so be patient!

Anyhow, if you plot `freq` versus `parameter1`, you'll see a nice relationship:

```
songs %>% filter(!is.na(pitch)) %>%
  mutate(as_music_df(pitch)) %>%
  ggplot(aes(freq,parameter1)) +
  geom_point()
```

In fact, if you add " + scale_x_log10()" to the end of the plot command, you'll get a linear relationship.

This answers the question as to what `parameter1` means: consecutive musical notes in equal tempered scales are not spaced evenly, they're spaced exponentially! Moreover, the spacing between adjacent pitches is a factor of the 12-th root of 2, so that there are 11 pitches in an octave. A MIDI file does not contain the value shown in the `pitch` column, nor does the file store frequency in Hz. Instead, it stores a note number: where 0 corresponds to 8.18 Hz, and 69 = 440 Hz = Concert A = 'a4'. So, `parameter1` is simply the note number (=pitch) as it was stored in the MIDI file.

10. Given what we learned in Step 9, the parameter2 is probably the raw value of something else in the MIDI file. That's true, in fact... the other thing that MIDI files store for each note is the note's "velocity" -- how loud it is. If you compare `parameter2` and `velocity`, you can see these values are the same. It's easy to verify this by plotting:

```
songs %>% filter(!is.na(pitch)) %>%
  ggplot(aes(x=velocity,y=parameter2)) + geom_point()
```

11. Now that we understand the meanings of the different columns, let's start to analyze the music in each of these files. The velocity (loudness) distribution for each file is a good place to start.

```
songs %>% group_by(name) %>% ggplot(aes(x=name,y=velocity)) + geom_boxplot()
```

Recall that ggplot knows about groups, and respects them when plotting. That's handy!

Right away, you can see that the duodecaphonic piece has no velocity variation at all. That's because the composer didn't specify it. The two improvised pieces have pretty wide variations in velocity. You can clearly hear that some notes are louder than others if you play those MIDI files. Additionally, although I don't know for sure, the Bach piece has quite a bit of variation as well. This probably means that the file was recorded from a live instrument, and then the resulting recording was "cleaned up" on a computer afterwards.

12. Similar to velocity, we can explore note lengths as well. Practically the same command works

```
songs %>% group_by(name) %>% ggplot(aes(x=name,y=length)) + geom_boxplot()
```

Notice that the duodecaphonic piece is again rather different from the others. If you remember in Step 3, we found that it had a mismatch between the "Note On" and "Note Off" events. Some of its notes are really long; that's consistent with the fact that some notes are held past the end of the piece!

13. That note length variability is important to a musician's "signature style". A boxplot is a too simplistic to see it. Let's look at the distribution of note lengths in more detail using a combination of plots we've used many times in this course:

```
songs %>% group_by(name) %>%
  count(length) %>%
  ggplot(aes(x=length,y=n,color=name)) +
  geom_point() +
  scale_x_log10() + scale_y_log10()
```

The log scales at the end are not necessary, but since most human senses work logarithmically anyway, it makes for a better plot.

Bach is rather more rigid from a tempo perspective, with a very large mode in note length. The improvisational pieces also have a clear tempo, as evidenced by the presence of a single mode. There is some jitter due to the fact that the improvisional piece was recorded with an (artificially) fast tempo. On the other hand, the duodecatonic piece is more spread out than Bach, but still has less jitter than the improvisation. Tempo detection from recordings is still an active area of research, because of all this jitter!

14. Let's delve a little further into this "Note On" versus "Note Off" business. Given the hypothesis that these are usually paired with each other, we can ask what role the "Note Off" events play. Let's just look at one file for the moment. Take a look at the times of the "Note Off" events:

```
note_off_times<-edwin_improv2 %>% filter(event=='Note Off') %>% select(time)
```

If you look at a typical "Note On" event, it has a time and a length. If we add these together (making a new column `note_end`), how do these compare to the "Note Off" times?

```
note_on_times_plus_length <- edwin_improv2 %>% filter(event=='Note On') %>%
  mutate(note_end=time+length) %>% # Make a new column...
  select(note_end) %>%  # Just display that new column
  arrange(note_end)   # Sort ascending, to match the "Note Off" results above
```

They are the same! You can check that this is true using

```
all(note_off_times == note_on_times_plus_length)
```

This means that we generally only need to worry about the "Note On" events.

Aside: When a MIDI instrument is playing, timing is critical since your ear can hear differences in timing well into the milliseconds range (or even microseconds in some cases). Scanning through the MIDI file takes time. Having "Note Off" events in present in the MIDI file means that the instrument doesn't have to search as far in the file to figure out when to stop playing a note, especially if the note is a long one. This gives an instrument a little extra time to keep up with the music. This was a really big deal in 1983, when MIDI was first standardized, since embedded computers were slow! It's still important for modern instruments playing complex pieces of music... But for us, sitting at a modern computer and running analysis without time pressure, the "Note Off" events are redundant.

15. Let's do a little visualization of the music itself. Let's make a new data frame built from what we've learned thus far:

```
songs_notes <- songs %>% filter(!is.na(pitch)) %>% # We only need events with pitches defined
  mutate(as_music_df(pitch), # Look up note frequencies
         end_time=time+length) # Note lengths determine the end time for each note
```

16. Now let's examine how note velocity and pitch are related. This can sometimes detect a melody/harmony split.

```
songs_notes %>% group_by(name) %>%
  ggplot(aes(x=freq,y=velocity,color=length)) +
  geom_point() +
  facet_wrap(~name)  # This helps keep the figure from getting too cluttered
```

In the improvised pieces, you can see a melody/harmony split. But there's not a clear distinction in the duodecaphonic piece.

17. Plotting one song is now easy using geom_linerange, which is a bunch of line segments. Notice that the note start and end times go into xmin and xmax to delineate the length of each line segment:

```
songs_notes%>%filter(name=='edwin_improv') %>%
  ggplot(aes(xmin=time,xmax=end_time,y=freq,color=velocity)) +
  geom_linerange() +
  scale_y_log10() # Log scale on frequencies, so the y axis looks more like a piano keyboard
```

On thing that should jump out at you is the harmony is slow-moving and basically happens below 250 Hz. The melody jumps around much more, and is higher pitched. In this particular piece, the melody was played with the right hand and the harmony with the left.

Try this for the other pieces, too. You'll notice that there are two distinct movements in the Bach piece!

18. In the improvised pieces there is a particular key that the pitches are all consistent with. This is less true for the Bach piece and not at all true for the duodecaphonic piece. Let's explore that! We can use tabr to identify when notes are consistent with a particular key, like we did in HW3A. Edwin (the pianist) informed me that he was intending to play in C major, so let's see if that's true:

```
edwin_improv2 %>% filter(event=='Note On') %>%
  select(pitch) %>%
  map(~is_diatonic(.x,key='c'))
```

This produces a list of TRUE (if the note was in C major) and FALSE (if the note wasn't). You can see that most notes are indeed consistent with C major.

19. As it happens, Edwin's harmony and melody follow the C major key a bit differently. Let's split melody and harmony. From the notes plotted in Step 17, it's clear that the melody is everything above 250 Hz.

```
edwin_improv_notes <- songs_notes %>% filter(name == 'edwin_improv')

edwin_improv_melody<-edwin_improv_notes%>%filter(freq>250)
```

and the harmony is everything below that

```
edwin_improv_harmony<-edwin_improv_notes%>%filter(freq<250)
```

We can check to make sure that we have just the harmony:

```
edwin_improv_harmony %>%
  ggplot(aes(xmin=time,xmax=endtime,y=freq,fill=velocity)) +
  geom_linerange()
```

Yup; looks good!

20. You can now check the melody and harmony separately, counting the number of notes that are consistent with that key:

```
edwin_improv_melody %>%
  select(pitch) %>%
  mutate(keycheck=map(pitch,~is_diatonic(.x,key='c'))) %>%
  count(keycheck==TRUE)
```

and

```
edwin_improv_harmony %>%
  select(pitch) %>%
  mutate(keycheck=map(pitch,~is_diatonic(.x,key='c'))) %>%
  count(keycheck==TRUE)
```

Apparently Edwin's harmony fits the key of C major rigidly, while is melody sometimes drifts.

21. (Optional, but worth it) If you're interested to see if there are any other keys that fit a particular piece, you can loop over the possible keys that tabr knows about, and count up the notes that do/do not fit each key. Because tabr isn't optimized for speed, this takes a long time on my computer (about 10 minutes per file), but the results are definitely interesting if you're willing to wait.

In order to do this, we basically want to run the same code as above, especially the map command. We will add a column to the data for each column, in which the column name matches the key. Helpfully, tabr can give you a list of keys with the keys() command.

Unfortunately, you want to both make a new column with each key **and** use it in the code. This is tricky because column names are not strings, but the tabr key names are strings. The way out is to "unquote" the key names with the "!!" (double exclamation point) operator. And, use a for loop. Here's the whole process for the melody we extracted in Step 19:

```
ei<-edwin_improv_melody # The ei table is a temporary table. That way if something goes wrong, we can backtrack!
for (k in keys())  {
  ei<-ei%>%
    mutate(!!k:=map(pitch,~is_diatonic(.x,key=k)))  # add the key as a new column, listing all notes and the test results
}
```

Once we've done that (WAIT!!!!), then we can re-pivot all those separate columns into a single column that's easier for analysis

```
ei<-ei %>% pivot_longer(cols=keys(),names_to='key')
```

And then we can plot the results:

```
ei%>%group_by(key) %>% # Grouping by keys to count note matches by key
  filter(value==TRUE) %>%  # Looking for notes that match the key in question
  count(value) %>%              # Count up those matching notes!
  ggplot(aes(reorder(key,desc(n)),n)) + # Sort by match count
  geom_col()
```

Interpretation: the larger values mean a better match for that key! In the case of Edwin's pieces, they're definitely C major. If you try the duodecaphonic piece, you'll find that it's not really in any particular key! Neat!

<u>Assignment</u>:

For submission, include your .R file as well as any MIDI files this file uses. Please include a citation for each such MIDI file (this can be a URL to where you downloaded that specific file).

1. Obtain two or more MIDI files (legally, please!) for your analysis. The files **must** come from at least two different sources. You can find many MIDI files online (see links on the first page; Wikipedia also has a number of good MIDI files that are legal to use), record your own, or compose your own. If you create your own file, it must have at least 100 notes in it.

You can make your own MIDI using an online sequencer

https://onlinesequencer.net/

Once you've played some music, you can Export to MIDI.

2. Load your MIDI files into RStudio, and explore the header information. Explain in a comment in your R file what MIDI events are present, which events are not present (if any). Try to explain why this is the case -- for instance, are either of the files recordings?

3. Plot the note length distribution for each of the files. Explain the features you see in a comment -- are all of the note lengths evenly distributed? Why or why not?

4. Create a plot of the notes velocity versus pitch (like Step 16). Explain what features are visible! Does this help you see a melody?

5. Create a plot of the notes' frequencies as a function of time (like Step 17). In a comment explain some of the interesting features you see in the piece. Do your pieces have a melody and harmony that is visible?

6. Select a key to see if your pieces fit that key. (If you're musically inclined, pick the most reasonable key for the piece, otherwise pick one at random.) Compute the percentage of notes that are consistent with that key in each file.

7. (Optional) Try to determine which key is being used via the analysis in Step 21. You may choose to use only the first 100 or so notes in your file if it's big and the process takes too long to run. Warn me in a comment if it takes more than 5 minutes on your computer to run.