# Wrangling and Visualizing Musical Data

**Project Description**

Apply data-wrangling and visualization tools from the tidyverse to musical data. Find the most common chords and chord progressions in a sample of pop/rock music from the 1950s-1990s, and compare the styles of different artists. This project assumes familiarity with standard TidyVerse tools for R, in particular the tibble data structure and the dplyr and ggplot2 packages. No specific musical knowledge is required, though it may give you ideas for further exploration of the dataset after completing the project.

Before taking on this project we recommend that you have completed the following courses:

* [**Introduction to the Tidyverse**](https://www.datacamp.com/courses/introduction-to-the-tidyverse)
* [**Data Visualization with ggplot2 (Part 2)**](https://www.datacamp.com/courses/data-visualization-with-ggplot2-2)

This project uses a parsed and cleaned version of the [**McGill Billboard Dataset**](http://ddmal.music.mcgill.ca/research/billboard), version 2.0 (CC0 license).

**Task 1: Instructions**

Read in the McGill Billboard chord dataset.

* Load in the dplyr, readr, and ggplot2 packages.
* Read in 'datasets/bb\_chords.csv' using read\_csv and assign it to bb.
* Display the first rows of bb.

Make sure to use read\_csv (with an *underscore*) to read in the data. The read.csv function, which is built into R, has a number of problems which the read\_csv function avoids.

**Good to know**

This project assumes familiarity with standard tidyverse tools for R like the dplyr, ggplot2 and the pipe operator (%>%). Before taking on this project we recommend that you have completed the following courses:

* [**Introduction to the Tidyverse**](https://www.datacamp.com/courses/introduction-to-the-tidyverse)
* [**Data Visualization with ggplot2 (Part 2)**](https://www.datacamp.com/courses/data-visualization-with-ggplot2-2)

RStudio has created some very helpful cheat sheets for working in the tidyverse, including two that will be helpful for this project: [**Data Wrangling**](https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf) and [**Data Visualization with ggplot2**](https://www.rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf). If you're a serious data wrangler, you might even print them out and laminate them!

Take Hint

**Task 2: Instructions**

Find the most common chords in the McGill Billboard Dataset.

* Count the number of occurrences of each raw chord type in the dataset (bb) using count(), and sort the results from most common (highest count) to least common (lowest count).
* Store the result in bb\_count.
* Display the 20 most common chords.

For readability (and to do things the tidyverse way!), try to write your code as a string of verb-based commands, one command per line, connected by %>%.

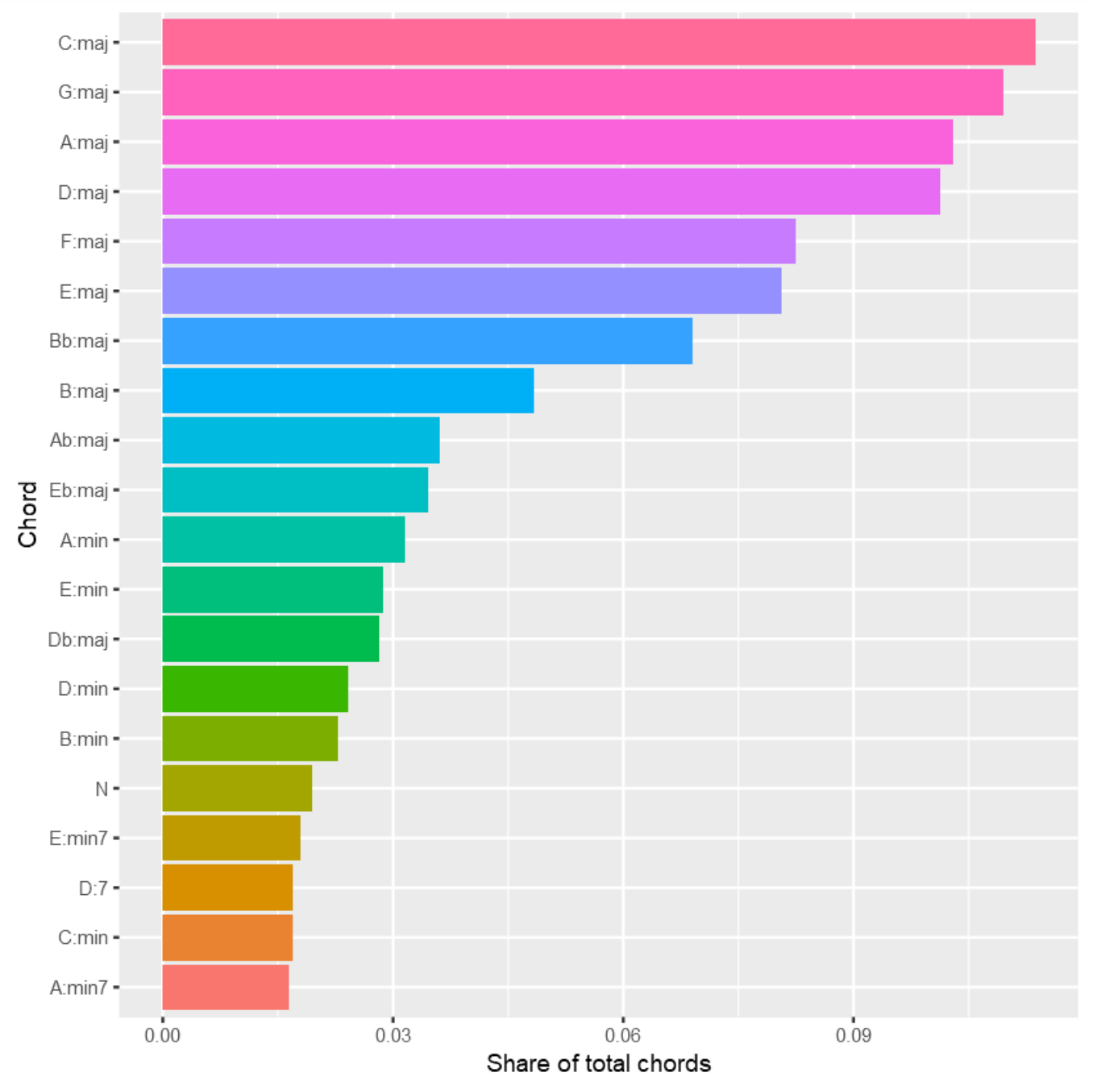
Take Hint

**Task 3: Instructions**

Plot the top 20 chords as a flipped bar plot.

* Starting with the first 20 records from bb\_count, use mutate to create a new column share with the percentage of how often each chord type occurs.
* Also using mutate, reorder the chord column according to the value in share.
* Pipe the results into ggplot() and make a column plot where the X axis represents chord and the Y axis is represents share.
* Make your plot more readable by adding labels with xlab() and ylab(), and by flipping the plot using coord\_flip().

Do your best to make your visualization look like this:

**[](https://projects.datacamp.com/projects/78)**

A picture is worth a thousand words -- perhaps, even more, when visualizing data! That's why we're working so hard to make the visualizations as readable as possible -- using percentages, arranging values in descending order, etc.

You may also try adding a splash of color. (Remember that column plots require color to be added with fill = chord rather than color.) When color adds to the aesthetic, but not a new dimension of information, I recommend removing the color legend with theme(legend.position='none').

As you're working through the above steps, think about what the plot would look like without some of these options. For example, what advantage does converting raw chords counts to percentages have for those reading the plot? How readable would the plot be without the axis labels? Without reordering columns? What value does coord\_flip() add to this plot?

Take Hint

**Task 4: Instructions**

Create a count of chord *bigrams*.

* Use mutate() to add two new columns to bb: next\_chord and next\_title. These should contain the data from the chord and title columns, but shifted one row up. Use the lead() function inside your mutate() command to do this.
* Create a bigram column that concatenates chord with next\_chord, with a space in between.
* Use filter() to remove any records in our new data frame where title and next\_title are not identical.
* Count the number of occurrences of each bigram type and store the results in bb\_bigram\_count.
* Display the 20 most common chord bigrams.

There are natural language processing (NLP) tools that will *tokenize* texts by *n-grams* (phrases of *n* words). However, our chord data is already in a tidy table, rather than in something that looks like paragraph form. Thankfully, dplyr contains functions like lag() and lead() that make it easy to access data from other rows in the data frame efficiently, and we can use them to construct our bigrams using paste() (or str\_c from stringr).

Why we filter in step 3 might not be obvious, but it's incredibly important. The last chord of one song combined with the first chord of the next song is *not* a bigram. Depending on the order of songs in the dataset, if we skip this step, we could end up with chord "progressions" connecting songs that occur perhaps 30 years apart in history!

Take Hint

**Task 5: Instructions**

Create a flipped bar plot that shows the 20 most common chord bigrams.

* Copy your code from Step 3, and modify it to work with bb\_bigram\_count instead of bb\_count.
* Adjust the plot labels to fit chord *changes* instead of just chords.

Copy-and-paste isn't cheating! In fact, knowing how to successfully copy, paste, and tweak existing code (yours, or someone else's -- with permission, of course) is an integral part of data science. It not only saves time and brain power, it also limits mistakes in your code when you use code you already know works. The iterative process of tweaking that code can also help you write more efficient code in the future.

Of course, if you copy-and-paste the same code several times, you may just want to write a custom function instead!

Take Hint

**Task 6: Instructions**

Find and display the 30 artists with the most songs in the McGill Billboard Dataset.

* Using bb, isolate the artist and title columns using select().
* We still have one record per *chord*. Use unique() to remove duplicates and leave a single record per *song*.
* As in earlier tasks, use count() to find how many songs each artist has in the dataset, and sort the results in descending order.
* Display the first 30 records in the sorted table.

In order to tag as many songs as possible quickly in the next task, we can simply identify a small number of prolific artists whose songs we can tag all at once. By isolating the 30 most prolific artists in the dataset, we can look at the results and pick a few good candidates.

When used in a piped string of commands, unique() does not need to take any arguments, since each command treats the output of the previous command as its first argument.

Take Hint

**Task 7: Instructions**

Add a new column instrument to bb, including "piano" or "guitar" for piano- and guitar-driven songs.

* Use inner\_join() with tags to attach an instrument column to bb and assign the result to bb\_tagged.
* Display the new data frame bb\_tagged to make sure the join was successful.

When adding a custom column to an entire data frame based on data in another column, it is usually much faster to use the appropriate join operation than to write a looping function. inner\_join() will even remove all rows in bb that do not correspond to the artists in tags. And in this case, since both bb and tags have an artist column, you do not need to specify a column by which to join.

Try it out with left\_join() and full\_join(), too. What are the differences? Do any produce the same results in this case? What would happen if you started with tags and applied a join operation to bb? Which join(s) would produce the desired results?

Take Hint

**Task 8: Instructions**

Created a faceted plot that shows the frequency of the most common chords side-by-side for songs by piano- and guitar-driven artists.

* Starting with bb\_tagged, use filter() to keep only the top\_20 chords.
* Use count() to find the number of times each chord occurs for each instrument, and sort the results.
* Pipe the results to ggplot() and make a bar plot, using chord as the X axis and n (the result of count()) as your Y axis.
* Use facet\_grid() to place guitar and piano plots side by side for comparison. Then use coord\_flip() for readability, and provide appropriate labels for the X and Y axes.

If you like, add a splash of color with fill (and if so, set theme(legend.position='none')).

facet\_wrap() and facet\_grid() are incredibly powerful visualization tools. They allow you to add dimensions to your data visualization story without making things hard on your readers.

Try playing around with faceting a bit. What happens when you count chord and artist and pass artist to facet\_grid()? What other parameters could you visualize in this way that tell a compelling story?

Take Hint

**Task 9: Instructions**

Create the same faceted plot as in Task 8, but for chord bigrams.

* Copy and modify your code from Task 4 to add a bigram column, this time to bb\_tagged.
* Copy and modify your code from Task 8 to produce a faceted plot of bigram frequency from the top\_20\_bigrams that compares guitar- and piano-driven songs.
* Remember to change all references to chords (including in the axis labels) to bigrams.

**HINT**

Use the same mutate and filter commands as in Task 4 to add a bigram column to bb\_tagged.

After that, the code will be almost identical to Task 8. Replace chord with bigram, and because we've already counted the bigram by instrument (code from Task 4), we do not need to do it again.

**Task 10: Instructions**

Complete the project by confirming the validity of the hypothesis, as well as the need for further data analysis to draw a conclusion.

* Is the hypothesis that guitar-driven and piano-driven songs have different chord tendencies valid and worth deeper exploration? TRUE or FALSE? Set hypothesis\_valid to reflect your answer.
* To draw a conclusion about this hypothesis, do we still need to explore more data? TRUE or FALSE? Set more\_data\_needed to reflect your answer.

Great work! You've uncovered some interesting things about musical chord progressions *and* learned a little about how natural language processing (NLP) analysis techniques can be used to study musical symbolic data.

**Do you want to know more?**

If this project got you hungry for more musical data analysis, check out my blog post, [**What is computational musicology?**](https://pushpullfork.com/computational-musicology/) There are links to academic studies, tools, and datasets for further exploration. The Million Song Dataset is especially cool.

And if you're also a Python fan, check out [**music21**](http://web.mit.edu/music21/), an advanced toolkit for computational musicology in Python.

Take Hint