

MEDB 5505, Module03

2025-02-01

Topics to be covered

- What you will learn
 - Reading text files
 - Comma delimited files
 - Tab delimited files
 - Other delimiters
 - Fixed width files
 - Real world examples
 - Your programming assignment

Text files, 1

- Advantages
 - Easy import into many programs
 - Review using notepad
- Disadvantages
 - Bigger size
 - Slower to import

Speaker notes

One of the most important skills you will learn in this class is how to read data from text files. Text files are commonly used for data storage because they are easily imported into a variety of different programs. You can often peek at a text file using a simple program like Notepad in order to get a quick feel for what the data looks like.

Text files are popular because just about any program out there can read a text file. You can import a non-text format, one that has special binary characters, but this often limits the types of programs that can read the data. So people who want to make their data widely available will almost store that data in a text file.

Text files do usually take more room than other formats. As cheap as storage is becoming, this may not be that serious a problem. You also need more computing time and power, but again this less of a problem as computers get faster and more powerful.

Text files, 2

- Wide range of formats
 - Delimited
 - Fixed width
- First row for variable names
 - Optional but recommended
- Always look for a data dictionary

Speaker notes

Text files come in a variety of format. A delimited file uses a special character, often a comma, to designate where one data value ends and the next one begins. In contrast, a fixed width file requires every variable to occupy a particular column or columns.

Many text files include the names of the variables in the first row of text. This is not required, but it is strongly recommended. I will talk about at least one example where you don't have the names of the variables in the first line of text.

Not every data set that you find is going to have a data dictionary but always look for it, because often that will give you some helpful advice. It will make it easier for you to read the data into a program like R.

Should I download before reading?

- Read directly from website
 - Convenient
 - Updates incorporated at each run
- Download then read
 - Downloaded file doesn't disappear
 - Avoid repeated long downloads
 - Work even when Internet connection is down

Speaker notes

R gives you the option of reading a file on your computer or reading it from a website. For small datasets that you only use once (such as for your homework assignments), it doesn't matter. For larger files and repeated data analyses, there are some advantages to reading directly from the website and some advantages to downloading the file to your computer.

Reading directly from the website is convenient. You don't have to figure out where to store your downloaded file. If the website updates the file on a regular basis, reading directly always insures that you have the most current data.

If you download the file and then read it, you provide yourself with some insurance against the website disappearing. If the download takes a long time, then you only have to endure that delay once. Finally, a downloaded file allows you to work when an Internet connection is not available, such as during a plane flight (though many airlines will now let you connect while in the air).

No data dictionary?

- Peek at file
 - Same number of delimiters on each line
 - Tabs versus multiple blanks are hard to distinguish

Speaker notes

Many files that you find on the Internet are missing any documentation or the documentation fails to help you figure out what approach to use to import the data. If that's the case, don't despair. There are several common sense things you can try.

First, peek at the file and see if there are any obvious delimiters. It's important that you have the exact same number of delimiters on each line of data. If you think the comma is the delimiter, then if there are five commas on one line, then every line should have five commas. The exception would be delimiters found inside quotes.

Tabs versus multiple blanks are hard to distinguish. This sometimes means that you will have difficulty telling whether to use a tab delimited file, a space delimited file, or a fixed width file. I have a web page that talks about this in detail.

No data dictionary?

- Experiment
 - Read warnings carefully
- If needed, edit the file manually
 - Simple edits of one or two offending lines
 - Global search and replace
 - Change tabs to blanks
 - Change multiple blanks to single blank

Speaker notes

There's nothing wrong with experimenting. Just pick one approach and try it. If you get an error message or the data is garbled, try a different approach. If you get warnings rather than errors, things may be okay, but look carefully at the warning message and double your vigilance efforts before you start analysis.

If all else fails, go in and edit the file manually. This helps if there are just a couple of rows in the file that are causing you heartburn. Usually the error or warning message will give you enough of a hint that you locate the offending lines. If there are problems on every line or almost every line, then sometimes a global search and replace works well.

Troubleshooting

- Some warning signs that it didn't work.
 - Multiple data read in as single variable.
 - Lots of missing values
 - Bottom looks different than top
 - `slice_head, slice_tail`

Speaker notes

Sometimes the effort to read in the data will seem like it works, but the data does not get read in properly. If you choose the wrong delimiter, then everything will get squished together into a single file.

You might see a lot of missing values, especially if R was looking for a numeric value for a particular variable, but the data set had string values in that location.

Also, peek at the bottom of the file and compare it to the top. Sometimes an import will get out of step partway through and every single row after that gets misaligned. Make sure that what you see at the bottom of dataset is roughly the same as what you see at the top.

Break #1

- What you have learned
 - Reading text files
- What's coming next
 - Comma delimited files

The readr library

- Part of tidyverse
- For importing text files
- Broad range of formats
- Very fast
- Makes intelligent guesses

Speaker notes

Although R has had a variety of functions for reading in text data, these were updated in the readr library, part of the tidyverse. These functions in readr import a broad range of text formats. The functions rely on C++ code and are very fast. They also make intelligent guesses that can often get your code up and running faster.

Useful functions in the readr library

- `read_csv()`: comma-separated values
- `read_tsv()`: tab-separated values
- `read_delim()`: arbitrary delimiter
- `read_fwf()`: fixed-width files
- `read_table()`: whitespace-separated files

Speaker notes

There are many functions in the readr library. Here are five of the more commonly used ones. The `read_csv` function will read text files where individual values are separated (delimited) by commas. The `read_tsv` function reads text files delimited by the tab character. Use the `read_delim` function to read files with anything other than a tab or comma as a delimiter. The `read_fwf` function reads files in a fixed width format. The `read_table` function reads files where values are separated by one or more whitespace characters (blanks and/or tabs).

You will see applications of these functions in this module.

Arguments for most readr functions

- `col_names =`
 - TRUE, FALSE, or a vector of names
- `col_types =`
 - “n” for numeric, “c” for character, “?” for guess
- `na =`
 - Defaults to “NA”
- `skip =`
 - How many rows to skip (defaults to 0)

Speaker notes

There are some arguments which you will see used in many of these readr functions. The `col_names` argument indicates whether the first row of the text file contains the names of each variable. It also allows you to specify names for the variables if they are not already provided. The `col_types` argument specifies which variables are numeric and which are strings. You can ask R to guess which is which by looking at the first few rows. The `na` argument allows you to specify how missing values are encoded in the text file. The default is NA, but you can use other codes. The dot (.) is a commonly used code for missing that is the default in SAS and SPSS. You can skip one or more rows of a file with the `skip` argument.

An example of a comma delimited file

```
x,y  
1,4  
2,8  
3,12  
4,16
```

Speaker notes

Here is a simple text file. It is just a file I made up to illustrate how to read text files. The first row provides the names of the two variables, x and y. Normally I do not recommend that you use such generic names as x and y.

The read_csv function

```
simple_comma <- read_csv(  
  file="../data/simple.csv",  
  col_names=TRUE,  
  col_types="nn")  
  
glimpse(simple_comma)
```


Live demonstration, 1

Now, you will see a live demonstration of part 1 of [simon-5505-03-demo](#).

Break #2

- What you have learned
 - Comma delimited files
- What's coming next
 - Tab delimited files

The evil tab character

- Jumps to a specific location
 - Location varies from program to program
- Looks like multiple blanks, but is a single character
- Can mask hidden blanks

Speaker notes

I do not like tabs. They cause all sorts of problems. The tab character is a way to imitate how manual typewriters (does anyone remember them?) would allow you to jump to a pre-specified location. This made it easy for a typist to create columns in a table.

The tab character can jump to a specific location, but that location varies from program to program. So a tab in one program might jump to columns 9, 17, 25, etc. and another might jump to columns 5, 9, 13, etc.

How to recognize a tab delimited file

- Partially aligned columns
- Everything is left justified

This is an example of a tab delimited file

Fat	Sodium	Calories
19	920	410
31	1500	580
34	1310	590
35	860	570
39	1180	640
39	940	680
43	1260	660

Speaker notes

I have a small data file on my github site, `burger-calories.txt`, that uses `tbs` as delimiters. Notice how the Sodium values are left justified. Also notice how some of the Calories values line up on column 9 and others line up on column 13.

This file is not tab delimited

Alpine	14.1	0.86	0.9853	13.6
Benson&Hedges	16.0	1.06	1.0938	16.6
BullDurham	29.8	2.03	1.1650	23.5
CamelLights	8.0	0.67	0.9280	10.2
Carlton	4.1	0.40	0.9462	5.4
Chesterfield	15.0	1.04	0.8885	15.0

Speaker notes

Here are the first few lines of another file on my github site, cigaretter-measurements.txt. Notice that the columns align perfectly. Also notice that the numbers are right justified. This is a white-space delimited file.

Tab delimited? Maybe, maybe not

9	1.7080	57.0	F	N
8	1.7240	67.5	F	N
7	1.7200	54.5	F	N
9	1.5580	53.0	M	N
9	1.8950	57.0	M	N
8	2.3360	61.0	F	N

Speaker notes

Here is a data set, where you can't be sure. This is a partial listing of fev.txt, also from my github site. Because everything in a column is exactly the same length, you can't distinguish between left and right justification. Actually, you can notice that all the data is right justified if you go down far enough. The point is, though, that sometimes you can't look at a file and recognize right away if it is tab delimited.

A simple tab delimited file

x	y
1	4
2	8
3	12
4	16

Speaker notes

Here is the same data, using tabs as delimiters.

Using the read_tsv function

```
simple_tab <- read_tsv(  
  file="../data/simple.tsv",  
  col_names=TRUE,  
  col_types="nn")  
  
glimpse(simple_tab)
```

Live demonstration, 2

Now, you will see a live demonstration of part 2 of [simon-5505-03-demo](#).

Break #3

- What you have learned
 - Tab delimited files
- What's coming next
 - Other delimiters

Anything can be a delimiter

$x \sim y$

1~4

2~8

3~12

4~16

Speaker notes

This file uses the tilde (~) as a separator (e.g., 1~4).

Why would you use a tilde as a delimiter? Sometimes your data itself includes delimiters like spaces and commas, and then you might want to choose an obscure out of the way symbol to serve as a delimiter.

Another obscure character that is sometimes used as a delimiter is the vertical bar (|).

Using the read_delim function

```
simple_tilde <- read_delim(  
  file="../data/tilde.txt",  
  delim="~",  
  col_names=TRUE,  
  col_types="nn")
```

```
glimpse(simple_tilde)
```

Live demonstration, 3

Now, you will see a live demonstration of part 3 of [simon-5505-03-demo](#).

Break #4

- What you have learned
 - Other delimiters
- What's coming next
 - Fixed width files

Reading fixed width format files

```
1 4  
2 8  
312  
416
```

Speaker notes

Here's another variation.

Sometimes you will get a file with no delimiters to save space. This requires that each variable takes up a fixed number of columns and that information is often specified in a separate file.

Okay, I deliberately did not put the variable names as the first line. Fixed width files often do not include the variable names in the file itself.

Disadvantages of fixed width formatting?

Speaker notes

When you have no delimiters, it is easy to get confused. The third line, for example, of our simple data file is “312”. That could be three numbers: 3, 1, and 2. It could be two numbers: 31 and 2? Or maybe 3 and 12? Or it could be a single number: 312. In a small dataset, you won’t get confused, but you might with a larger dataset.

It’s also more work because you have to specify the number of columns that each variable uses. That’s not trivial for a large file.

The fixed width format is also more prone to errors. If you get the columns wrong, you might truncate some of your data.

Example where fixed width formatting is needed.

C:\Users\steve\AppData\Local\Temp\Temp1_ED2017.zip\ED2017 - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

ED2017

```
16696 07714510053 0544-07011022011102-700100000002099208501813506809903-80215250153002205061100-0009C ^
16697 07307440088 0766-07011022011102-701100000002097007501813507909502-8021020012201411502370011000C
16698 07320500028 0494-07012022022202-7000001000050970082018112071098040002575501870218350-0009-0009C
16699 07611070013 0001124012022022202-7000100000030997155038-09-091000400021860028000-0009-0009-0009C
16700 07318320004 0333-07012022022202-70001000000309901160181330820970403021910155750581004115046050C
16701 07722330062 0303-07012022011102-700010000003098611002019810309903-802103512705045250-0009-0009C
16702 07615020000 0242-07012011-913010201000000001097808701811105309802-80259150153004115015250-0009C
16703 08122340000 0242-07011022011102-70001000000309820790181190800980200021165047351540505425054200C
16704 07101470007 0263-07012022011102-700000100005098408001810305510004-802191014115058100-0009-0009C
16705 07703280008 0242-07012022011102-70100000000109700870181460871000310021055210550171511525015300C
16706 08117070023 0363-07011022011102-700010000003097610002011308009903080210550191012705016750-0009C
16707 07412300096 0403-07012022011102-700100000002097108902014508409703030210553154534200041150-0009C
16708 07102350033 0756-07011022011102-70110000000209710760181310691000309021210010601-0009-0009-0009C
16709 07308520003 0353-070120220111010100001000004096506301814009209803090250250581001910119255-0009C
16710
```

Normal text file length : 41,037,304 lines : 16,710 Ln : 16,709 Col : 1 Sel : 0 | 0 Windows (CR LF) UTF-8 INS

Speaker notes

Here's practical example where you really might need fixed width formatting. This is data from Emergency Department visits in a CDC survey. There are over 16,000 rows, which is bad enough. But each row (except the last one) has over 2,400 characters. These columns contain information for over 900 variables.

If you were to use delimiters in this file, you'd have to add 900 commas or 900 tabs or 900 spaces or 900 tildes to each and every line. That works out to be 1.4 million commas across the entire file. That's a substantial increase to the size of an already very large and unwieldy file.

Helpful functions with read_fwf

- `fwf_empty()`
 - Uses spacing to guess at column positions
- `fwf_widths()`
 - Specifies column widths
- `fwf_positions()`
 - Specifies start and end locations for each column

Speaker notes

There are several ways to identify the column positions for a fixed width file. The `fwf_empty` function will make a reasonable guess as long as there is whitespace (one or more blanks/tabs) between each variable. You could also specify the column widths or the start and end location for each column.

The read_fwf function

```
simple_fixed <- read_fwf(  
  file="../data/fixed.txt",  
  col_types="nn",  
  col_positions = fwf_widths(  
    c(1, 2),  
    col_names=c("x", "y")))  
  
glimpse(simple_fixed)
```

Live demonstration, 4

Now, you will see a live demonstration of part 4 of [simon-5505-03-demo](#).

Break #5

- What you have learned
 - Fixed width files
- What's coming next
 - Real world examples

Function arguments for advanced options

- `col_select=`
- `na=`
- `name_repair=`
- `skip=`

Speaker notes

There are some nice options that I won't show in this video, but they are still worth noting.

You can use the `col_select` argument to read in only some of the columns of data.

Use the `na` argument to designate codes for missing values. By default, R looks for NA or a width of zero. But sometimes, other symbols, such as an asterisk or a dot may represent missing values.

You can use the `name_repair` argument to suggest how to handle names that are duplicates or which violate the rules in R for variable names.

You can use the `skip` argument to ignore a certain number of rows before reading data.

Example 1, binary.csv

Logit Regression | R Data Analysis

https://stats.oarc.ucla.edu/r/dae/logit-regression/

UCLA Advanced Research Computing
Statistical Methods and Data Analytics

Search this website

HOME SOFTWARE RESOURCES SERVICES ABOUT US

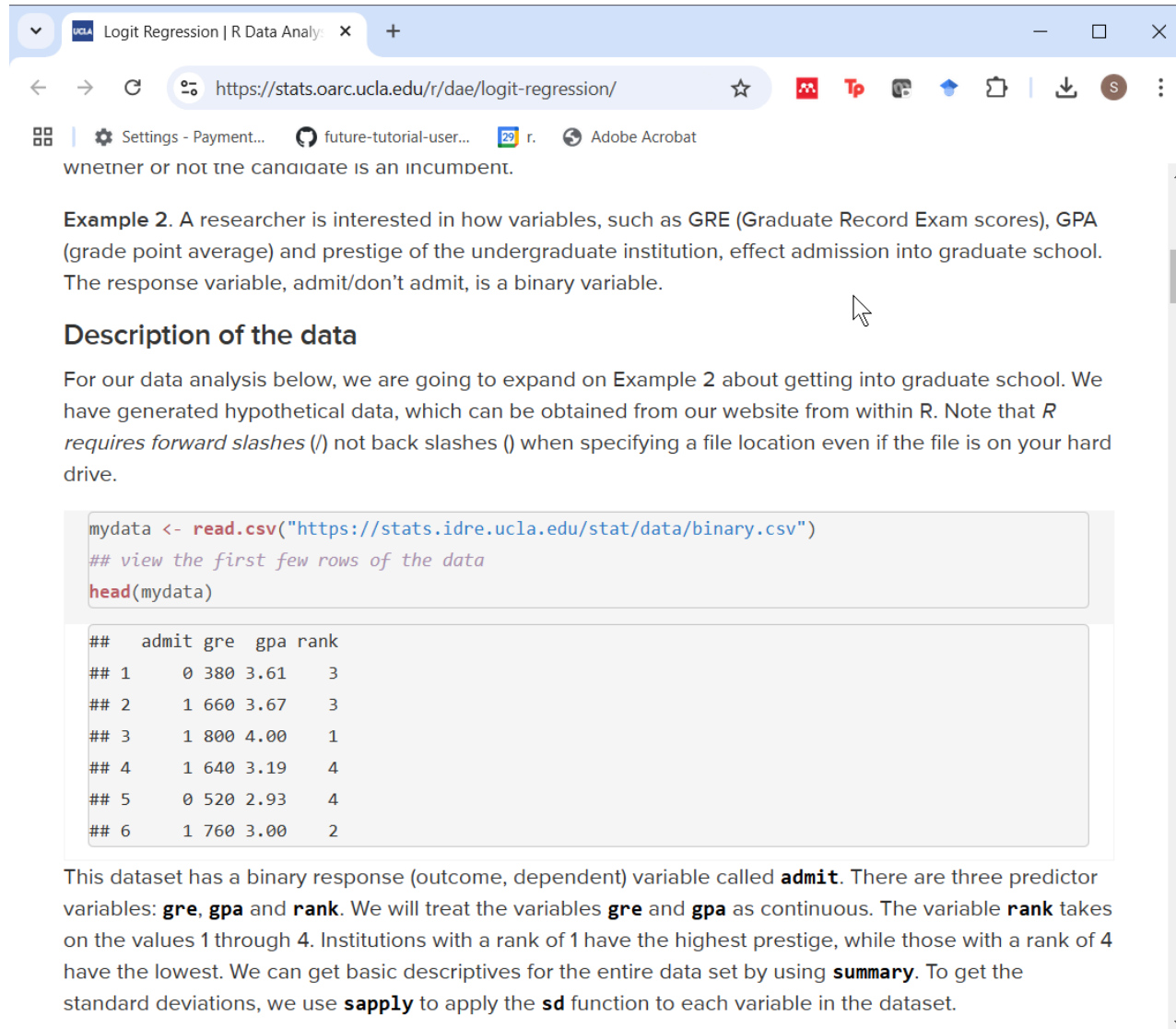
LOGIT REGRESSION | R DATA ANALYSIS EXAMPLES

Logistic regression, also called a logit model, is used to model dichotomous outcome variables. In the logit model the log odds of the outcome is modeled as a linear combination of the predictor variables.

Speaker notes

Let's look at some real world examples. The first one comes from one of the pages at the UCLA site on data analysis examples.

Example 1, a brief description



Logit Regression | R Data Analy x +

https://stats.oarc.ucla.edu/r/dae/logit-regression/

Settings - Payment... future-tutorial-user... r. Adobe Acrobat

whether or not the candidate is an incumbent.

Example 2. A researcher is interested in how variables, such as GRE (Graduate Record Exam scores), GPA (grade point average) and prestige of the undergraduate institution, effect admission into graduate school. The response variable, admit/don't admit, is a binary variable.

Description of the data

For our data analysis below, we are going to expand on Example 2 about getting into graduate school. We have generated hypothetical data, which can be obtained from our website from within R. Note that *R* requires forward slashes (/) not back slashes () when specifying a file location even if the file is on your hard drive.

```
mydata <- read.csv("https://stats.idre.ucla.edu/stat/data/binary.csv")
## view the first few rows of the data
head(mydata)
```

##	admit	gre	gpa	rank
## 1	0	380	3.61	3
## 2	1	660	3.67	3
## 3	1	800	4.00	1
## 4	1	640	3.19	4
## 5	0	520	2.93	4
## 6	1	760	3.00	2

This dataset has a binary response (outcome, dependent) variable called **admit**. There are three predictor variables: **gre**, **gpa** and **rank**. We will treat the variables **gre** and **gpa** as continuous. The variable **rank** takes on the values 1 through 4. Institutions with a rank of 1 have the highest prestige, while those with a rank of 4 have the lowest. We can get basic descriptives for the entire data set by using **summary**. To get the standard deviations, we use **sapply** to apply the **sd** function to each variable in the dataset.

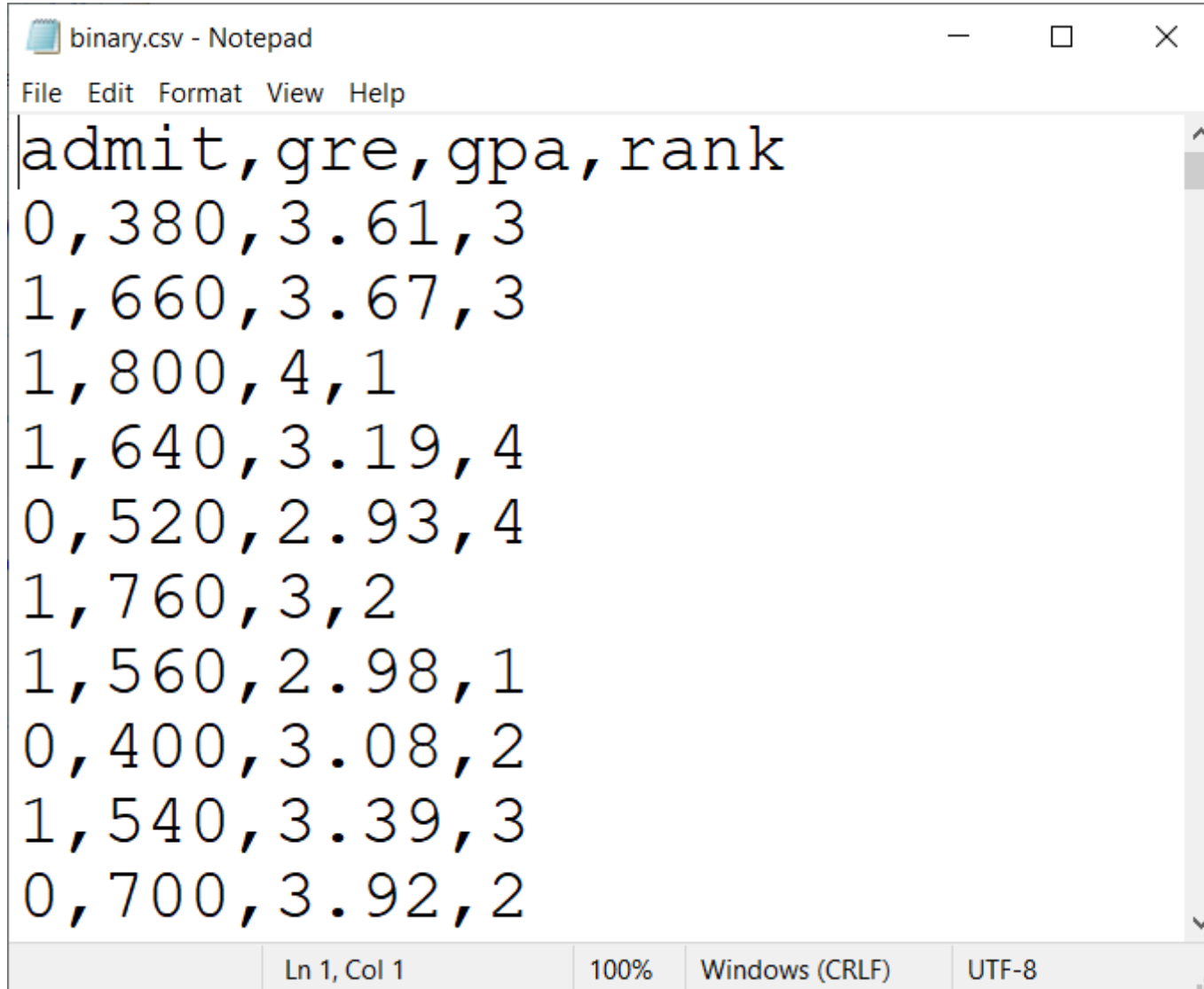
Speaker notes

This data set does not have a formal data dictionary, but there is pretty good description at their website

“This dataset has a binary response (outcome, dependent) variable called admit. There are three predictor variables: gre, gpa and rank. We will treat the variables gre and gpa as continuous. The variable rank takes on the values 1 through 4. Institutions with a rank of 1 have the highest prestige, while those with a rank of 4 have the lowest.”

Most people would describe the variable rank as ordinal. How you handle ordinal data in a formal data analysis is beyond the scope of this class, but let me just say that it is controversial. There is no consensus in the research community on how to handle ordinal data.

Example 1, viewing the file in Notepad



```
binary.csv - Notepad
File Edit Format View Help
admit,gre,gpa,rank
0,380,3.61,3
1,660,3.67,3
1,800,4,1
1,640,3.19,4
0,520,2.93,4
1,760,3,2
1,560,2.98,1
0,400,3.08,2
1,540,3.39,3
0,700,3.92,2
Ln 1, Col 1 100% Windows (CRLF) UTF-8
```

Speaker notes

I downloaded the file and peeked at it using the Notepad program in Microsoft Windows. If you are using a Mac, you might use TextEdit instead. For really big files or files that you need to make complex changes to, you might want to use a more advanced text editing program. I am partial to Notepad++ but there are many other programs that are just as good.

As a quick note, while I like to look at a file in Notepad or Notepad++, but I prefer to do any modifications inside of the R program itself. If you modify a text file outside of R, you should leave detailed notes about what changes you made so that others can reproduce your work.

Example 1, the code to peek at the data

```
[1] "admit,gre,gpa,rank" "0,380,3.61,3"      "1,660,3.67,3"  
[4] "1,800,4,1"          "1,640,3.19,4"      "0,520,2.93,4"  
[7] "1,760,3,2"          "1,560,2.98,1"      "0,400,3.08,2"  
[10] "1,540,3.39,3"
```

Speaker notes

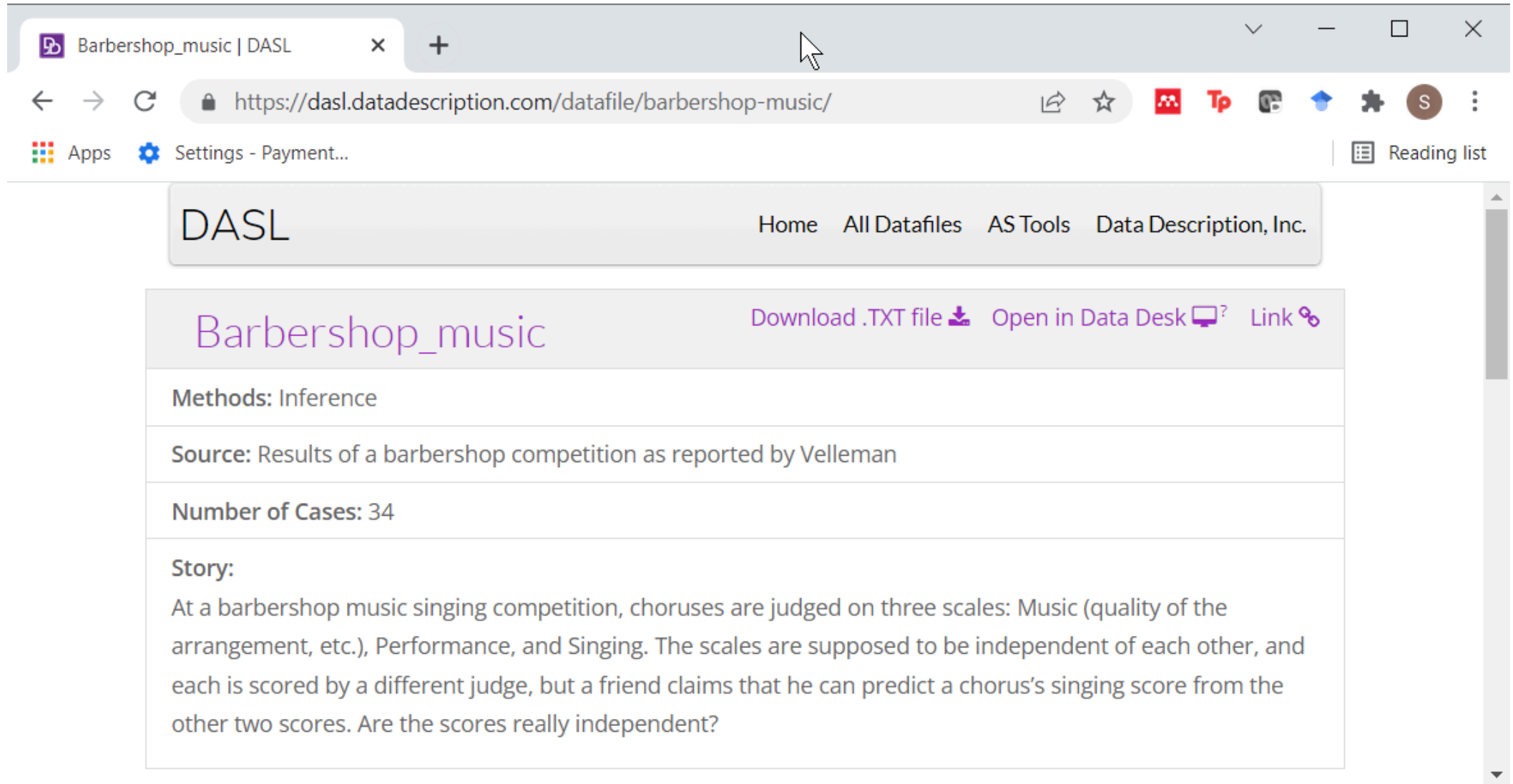
It is not needed for this small file, but I have also found that peeking at a dataset using the `read_lines` function can be helpful. Notice that I deliberately limit the number of lines read in because no one wants to look at hundreds of lines of data.

Always, always, always limit the amount of data that you display. The first time I taught the Introduction to SAS class, I had a student that submitted a pdf file of SAS output that was over a hundred pages long. I don't mind looking at a hundred pages of output. I am very good at skimming and jumping around. But this is not something you should do to your colleagues. Many of them will get really annoyed at you.

Example 1, the code to read the data

```
example_1 <- read_csv(  
  file=url_1,  
  col_names=TRUE,  
  col_types="nnnn")  
  
glimpse(example_1)
```

Example 2, barbershop-music.txt



The screenshot shows a web browser window with the address bar displaying `https://dasl.datadescription.com/datafile/barbershop-music/`. The browser's address bar includes navigation icons (back, forward, refresh), a lock icon, and a star icon. The browser's toolbar shows various extension icons (e.g., Apps, Settings, Payment, Reading list) and a search icon. The DASL website header features the DASL logo and navigation links: Home, All Datafiles, AS Tools, and Data Description, Inc. The main content area displays the datafile name 'Barbershop_music' in purple, followed by links for 'Download .TXT file' (with a download icon), 'Open in Data Desk' (with a Data Desk icon), and 'Link' (with a link icon). Below this, the datafile details are presented in a table-like structure:

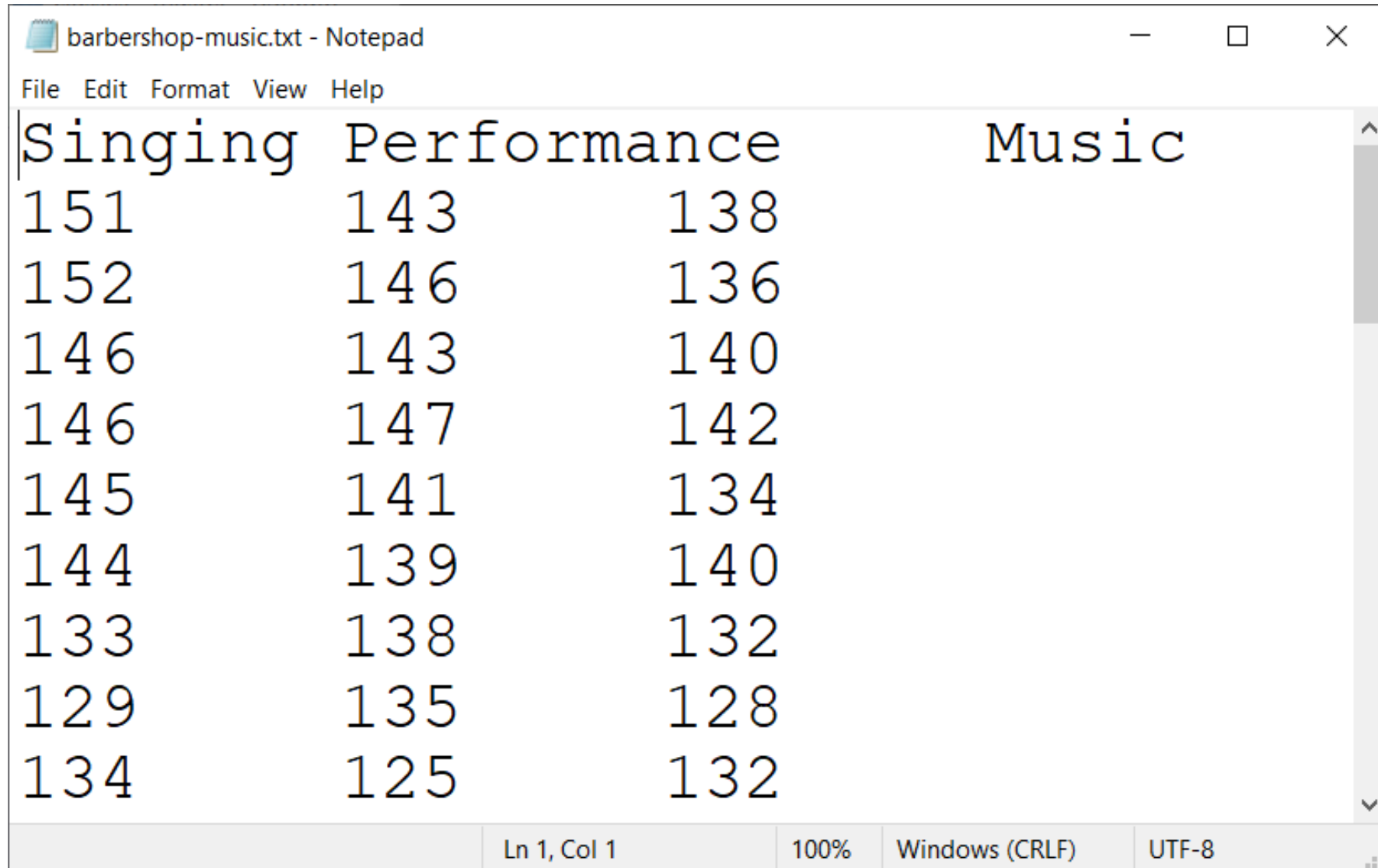
Methods: Inference
Source: Results of a barbershop competition as reported by Velleman
Number of Cases: 34
Story: At a barbershop music singing competition, choruses are judged on three scales: Music (quality of the arrangement, etc.), Performance, and Singing. The scales are supposed to be independent of each other, and each is scored by a different judge, but a friend claims that he can predict a chorus's singing score from the other two scores. Are the scores really independent?

Speaker notes

Here is a second dataset, barbershop-music.txt, from the DASL (Data and Story Library) repository. There is no formal data dictionary, but the description on the website is somewhat helpful.

“At a barbershop music singing competition, choruses are judged on three scales: Music (quality of the arrangement, etc.), Performance, and Singing. The scales are supposed to be independent of each other, and each is scored by a different judge, but a friend claims that he can predict a chorus’s singing score from the other two scores. Are the scores really independent?”

Example 2, viewing the file in Notepad



barbershop-music.txt - Notepad

File Edit Format View Help

Singing	Performance	Music
151	143	138
152	146	136
146	143	140
146	147	142
145	141	134
144	139	140
133	138	132
129	135	128
134	125	132

Ln 1, Col 1 100% Windows (CRLF) UTF-8

Speaker notes

Looking at it in notepad, it looks like it is a tab delimited file. It might be a fixed width file, but it takes a careful eye to spot clues to why this is not a fixed width file. A few carefully chosen test edits to the file will confirm that this is indeed a tab delimited file.

Example 2, peeking at the data

```
[1] "Singing\tPerformance\tMusic" "151\t143\t138"  
[3] "152\t146\t136"                "146\t143\t140"  
[5] "146\t147\t142"                "145\t141\t134"  
[7] "144\t139\t140"                "133\t138\t132"  
[9] "129\t135\t128"                "134\t125\t132"
```

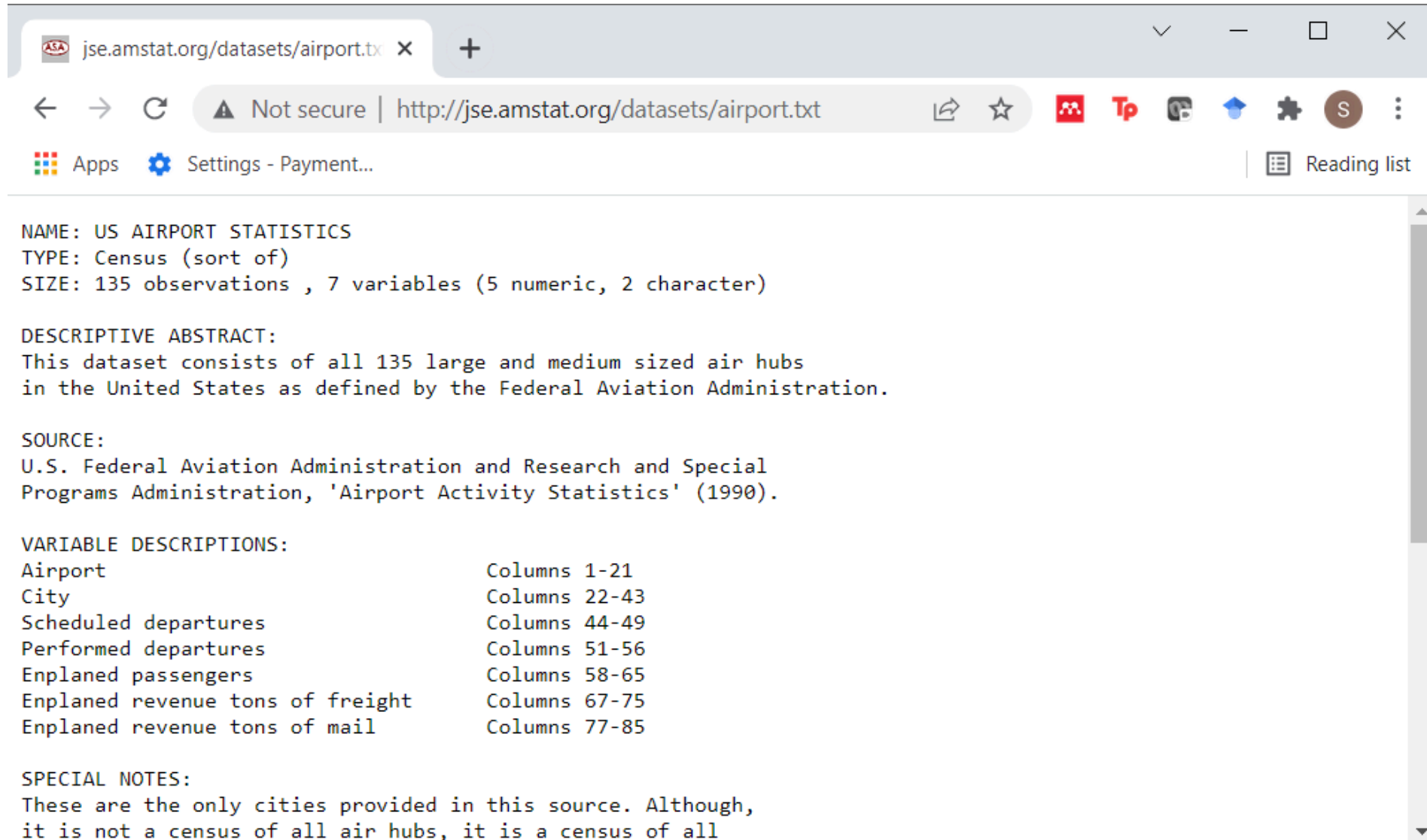

Speaker notes

When you use the `read_lines` function to peek at the data, notice how R displays the tabs using a special code `\t`. This is a common shorthand used by many programs. Other programs use `^t`.

Example 2, the code to read the data

```
example_2 <- read_tsv(  
  file=url_2,  
  col_names=TRUE,  
  col_types="nnn")  
  
glimpse(example_2)
```

Example 3, airport.txt



The screenshot shows a web browser window with the address bar displaying 'http://jse.amstat.org/datasets/airport.txt'. The page content is as follows:

NAME: US AIRPORT STATISTICS
TYPE: Census (sort of)
SIZE: 135 observations , 7 variables (5 numeric, 2 character)

DESCRIPTIVE ABSTRACT:
This dataset consists of all 135 large and medium sized air hubs in the United States as defined by the Federal Aviation Administration.

SOURCE:
U.S. Federal Aviation Administration and Research and Special Programs Administration, 'Airport Activity Statistics' (1990).

VARIABLE DESCRIPTIONS:

Airport	Columns 1-21
City	Columns 22-43
Scheduled departures	Columns 44-49
Performed departures	Columns 51-56
Enplaned passengers	Columns 58-65
Enplaned revenue tons of freight	Columns 67-75
Enplaned revenue tons of mail	Columns 77-85

SPECIAL NOTES:
These are the only cities provided in this source. Although, it is not a census of all air hubs, it is a census of all

Speaker notes

The third example comes from the Journal of Statistics Education website.

Example 3, peeking at the file on the web

Speaker notes

This is a third data set. Notice that there are no variable names at the top of the file.

Example 3, a description of the data

- Here is an excerpt from the data dictionary.

VARIABLE DESCRIPTIONS:

Airport	Columns 1-21
City	Columns 22-43
Scheduled departures	Columns 44-49
Performed departures	Columns 51-56
Enplaned passengers	Columns 58-65
Enplaned revenue tons of freight	Columns 67-75
Enplaned revenue tons of mail	Columns 77-85

Speaker notes

From the description on the website, you can tell that it is a fixed width format.

Example 3, the code to peek at the data

```
url_3 <- "http://jse.amstat.org/datasets/airport.dat.txt"
read_lines(
  file=url_3,
  n_max=10)
```

Speaker notes

You could also infer that this is a fixed width file using the `read_lines` function.

Example 3, Defining variable names and column locations

```
start_column <- c( 1, 22, 44, 51, 58, 67, 77)
end_column <-   c(21, 43, 49, 56, 65, 75, 85)
variable_names <- c(
  "airport",
  "city",
  "scheduled_departures",
  "performed_departures",
  "enplaned_passengers",
  "enplaned_freight",
  "enplaned_mail")
```

Speaker notes

You have to specify a bit of information before using the `read_fwf` function.

Example 3, the code to read the data

```
example_3 <- read_fwf(  
  file=url_3,  
  col_types="ccnnnnnn",  
  col_positions=fwf_positions(  
    start=start_column,  
    end=end_column,  
    col_names=variable_names))  
  
glimpse(example_3)
```

Speaker notes

You could use the `fwf_widths` in R. Just do a bit of arithmetic. The first variable goes from columns 1 to 21, it has a width of 21. The second variable goes from columns 22 to 43, so it has a width of 22. And so forth. The `fwf_positions` function is a bit more verbose, but less error prone because R does the basic math for you.

Break #6

- What you have learned
 - Real world examples
- What's coming next
 - Your programming assignment

This programming assignment was written by Steve Simon on 2024-12-18 and is placed in the public domain.

Program

- Create a single program to address the questions below.
 - Refer to the module 03 demonstration programs as needed.
 - Store your program in the src folder
 - Follow the naming conventions recommended for this class
 - Include the appropriate documentation

Question 1

The oyster dataset shows two different computer vision methods that can be used to estimate oyster weight and oyster volume. Please consult the [data description](#) and then review [the dataset itself](#). This is a tab delimited file. Read in the file and show a glimpse of the data. No interpretation of the output is needed.

Question 2

The file `diamond.txt` is data from a study of diamond ring prices. Please consult the [data description](#) and then review [the dataset itself](#). This is a fixed width text file. Read in the file and show a glimpse of the data. No interpretation of the output is needed.

Grading rubric

You will be evaluated using the [general grading rubric for programming assignments](#).

Your submission

- Save the output in html format
- Convert it to pdf format.
- Make sure that the pdf filename includes
 - Your last name
 - The number of this course
 - The number of this module
- Upload the file

If it doesn't work

Please review the [suggestions](#) if you encounter an error page.

Summary

- What you have learned
 - Reading text files
 - Comma delimited files
 - Tab delimited files
 - Other delimiters
 - Fixed width files
 - Real world examples
 - Your programming assignment

