post02-armaan-kohli

Armaan Kohli

11/22/2017

Factoring in R

Introduction

Factors are a very important tool to work with in R when it comes to working with categorical variables. They will help us in a variety of ways when it comes to analyzing our data as it lets us aggregate different categories together, reorder a table based on a value or a feature corresponding to a factor (that's not alphebetical!), and more.

Before we start working on anything, let's load the necessary packages that we want to work with so that we can look at the value of factors. If you don't have the forcats and tidyverse packages, you can install them first with the install.packages function like this below:

```
#install.packages('tidyverse')
#install.packages('forcats')
```

Now, just get them with the library function.

```
library(tidyverse)
```

```
## Loading tidyverse: ggplot2
## Loading tidyverse: tibble
## Loading tidyverse: tidyr
## Loading tidyverse: readr
## Loading tidyverse: purrr
## Loading tidyverse: dplyr
```

```
## Warning: package 'dplyr' was built under R version 3.4.2
```

```
## Conflicts with tidy packages ----------------------------------------------
```

```
## filter(): dplyr, stats
## lag():    dplyr, stats
```

```
library(forcats)
```

The Basics

Creating a Factor

Let's look at a quick example of a random basketball team. Here is a vector of strings, as of right now, that represent the different positions that each player plays:

```
positions <- c("PG", "SF", "SG", "PF", "SF", "SG", "PF",
"C", "C", "SF", "SF", "PG", "PF", "PF", "PG")
positions
```

```
##  [1] "PG" "SF" "SG" "PF" "SF" "SG" "PF" "C"  "C"  "SF" "SF" "PG" "PF" "PF"
## [15] "PG"
```

Now, we will convert this vector of strings into a factor using the 'factor' function.

```
positions_factor <- factor(positions)
positions_factor
```

```
##  [1] PG SF SG PF SF SG PF C  C  SF SF PG PF PF PG
## Levels: C PF PG SF SG
```

Let's note some differences in the printouts: 1. There are no more double quotes around each item. 2. Factor levels for each of the corresponding categories are also printed.

The factor function goes through the vector and looks to see at all of the different categories present (in this example, it's "C PF PG SF SG" because R sorts the values alphabetically) and then converts the vector to a vector of integers. Calling structure makes this more clear:

```
str(positions_factor)
```

```
##  Factor w/ 5 levels "C","PF","PG",..: 3 4 5 2 4 5 2 1 1 4 ...
```

"C" corresponds to 1, "PF" corresponds to 2, "PG" corresponds to 3, "SF" corresponds to 4, "SG" corresponds to 5. Essentially, factors are just integer vectors with integers correspondding to a certain category.

Giving Levels to a Factor

If you want the ordering of the factors to be different, there is a way to do that! For example, in basketball, Point Guards are typically called position 1, shooting guards called position 2, small forwards called position 3, power forwards called position 4, and centers called position 5. So, we can order it just like that!

```
positions_factor2 <- factor(positions, levels = c("PG", "SG", "SF", "PF", "C"))
str(positions_factor2)
```

```
##  Factor w/ 5 levels "PG","SG","SF",..: 1 3 2 4 3 2 4 5 5 3 ...
```

```
positions_factor2
```

```
##  [1] PG SF SG PF SF SG PF C  C  SF SF PG PF PF PG
## Levels: PG SG SF PF C
```

Renaming Levels and having Labels

Sometimes, you may want to rename your levels and have corresponding labels for your factor. In our example, we may want to write out the entire position instead of its abbreviation.

```
levels(positions_factor2) <- c("PointGuard", "ShootingGuard", "SmallForward", "PowerForward", "Center")
positions_factor2
```

```
##  [1] PointGuard    SmallForward  ShootingGuard PowerForward  SmallForward
##  [6] ShootingGuard PowerForward  Center        Center        SmallForward
## [11] SmallForward  PointGuard    PowerForward  PowerForward  PointGuard
## Levels: PointGuard ShootingGuard SmallForward PowerForward Center
```

We can also use the labels argument in the factor function to specify category names, but note how I have to specify the labels in alphabetic order, which is a pain.

```
factor(positions, labels = c("Center", "PowerForward", "PointGuard", "ShootingGuard", "Small Forward"))
```

```
##  [1] PointGuard    ShootingGuard Small Forward PowerForward  ShootingGuard
##  [6] Small Forward PowerForward  Center        Center        ShootingGuard
## [11] ShootingGuard PointGuard    PowerForward  PowerForward  PointGuard
## Levels: Center PowerForward PointGuard ShootingGuard Small Forward
```

However, as you may have noticed, specifying labels and levels separately can be confusing. In the same factor function, we can specify the levels to get the desired order, and them we can specify the new names for the factor with the labels.

```
positions_factor3 <- factor(positions, levels = c("PG", "SG", "SF", "PF", "C"),
       labels = c("PointGuard", "ShootingGuard", "SmallForward", "PowerForward",
                  "Center"))
positions_factor3
```

```
##  [1] PointGuard    SmallForward  ShootingGuard PowerForward  SmallForward
##  [6] ShootingGuard PowerForward  Center        Center        SmallForward
## [11] SmallForward  PointGuard    PowerForward  PowerForward  PointGuard
## Levels: PointGuard ShootingGuard SmallForward PowerForward Center
```

Now, whenever I look at my factor, everything is in the right order and it has proper labels.

Checking Frequencies

Also, now that we have converted to factors, we can check the frequencies of each position in a few different ways.

```
table(positions_factor3)
```

```
## positions_factor3
##    PointGuard ShootingGuard  SmallForward  PowerForward        Center
##             3             2             4             4             2
```

```
fct_count(positions_factor3)
```

```
## # A tibble: 5 x 2
##               f     n
##          <fctr> <int>
## 1    PointGuard     3
## 2 ShootingGuard     2
## 3  SmallForward     4
## 4  PowerForward     4
## 5        Center     2
```

Both of these work well with factors and allow us to see the frequencies of each factor.

Notice, that working with just the character vector, we receive a meaningless output.

```
table(positions)
```

```
## positions
##  C PF PG SF SG
##  2  4  3  4  2
```

Ordering Factors

There are nominal categorical variables, which don't have a distinct order, and ordinal variables, which do have a distinct order. It makes sense to give some lists a distinct ordering whereas it doesn't make much sense to give others an ordering. For example, we can't really say that shooting guards are less than small forwards. This will show an error:

```
positions_factor3[2] < positions_factor3[3]
```

```
## Warning in Ops.factor(positions_factor3[2], positions_factor3[3]): '<' not
## meaningful for factors
```

```
## [1] NA
```

However, there are examples in which ordering should matter. Let's just say we have a bunch of items on a to-do list and we have given them a priority. We want something with a HIGH priority to be greater than something with LOW or MEDIUM priority.

```
priority <- c("Low", "Medium", "High", "High", "Medium", "Low", "Low")
priority_factor <- factor(priority, levels = c("Low", "Medium", "High"), labels = c("Lo", "Me", "Hi"), ordered = T
RUE)
str(priority_factor)
```

```
##  Ord.factor w/ 3 levels "Lo"<"Me"<"Hi": 1 2 3 3 2 1 1
```

Now, we can see that something with a low priority is less than something with a high priority and something with a medium priority is not greater than something with a high priority.

```
priority_factor[1] < priority_factor[3]
```

```
## [1] TRUE
```

```
priority_factor[2] > priority_factor[3]
```

```
## [1] FALSE
```

Practical Application

To get a better feel for the factor, we are going to be working a lot with the dataset in the forcats package, gss_cat.

So, for ease purposes, look at the dataset in the pane by calling View on it and try to get an understanding of what the data is showing us.

```
gss_cat
```

```
## # A tibble: 21,483 x 9
##     year       marital   age   race        rincome            partyid
##    <int>        <fctr> <int> <fctr>         <fctr>             <fctr>
##  1  2000 Never married    26  White  $8000 to 9999       Ind,near rep
##  2  2000      Divorced    48  White  $8000 to 9999 Not str republican
##  3  2000       Widowed    67  White Not applicable        Independent
##  4  2000 Never married    39  White Not applicable       Ind,near rep
##  5  2000      Divorced    25  White Not applicable   Not str democrat
##  6  2000       Married    25  White $20000 - 24999    Strong democrat
##  7  2000 Never married    36  White $25000 or more Not str republican
##  8  2000      Divorced    44  White  $7000 to 7999       Ind,near dem
##  9  2000       Married    44  White $25000 or more   Not str democrat
## 10  2000       Married    47  White $25000 or more  Strong republican
## # ... with 21,473 more rows, and 3 more variables: relig <fctr>,
## #   denom <fctr>, tvhours <int>
```

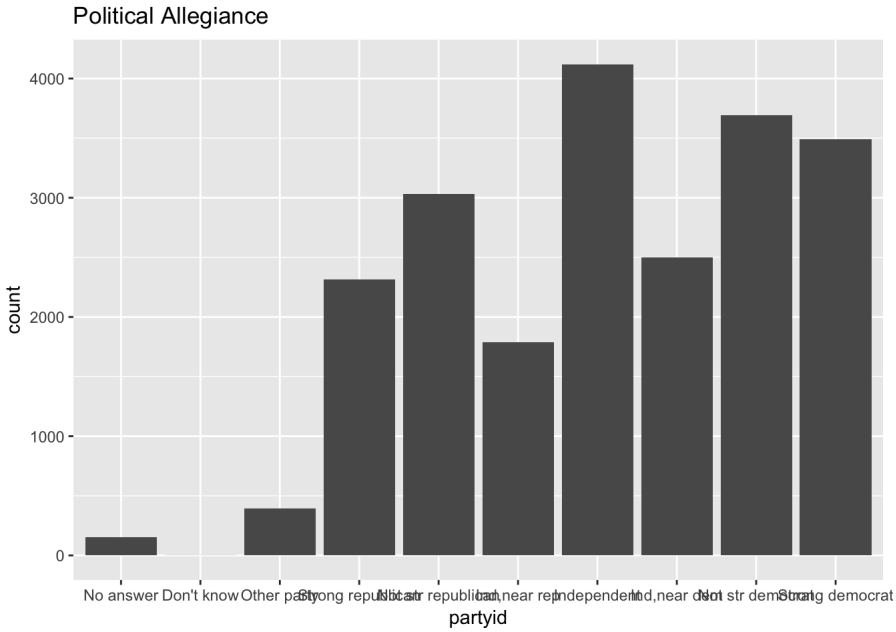
```
#View(gss_cat)
```

This was a survey in which the year a person answered this was recorded along with their age, race, income, party, religion, denomination, and hours they spend watching TV a day. As you can see there are a lot of categories for each answer and that some of them are similar and can be grouped together. We will be using the power of factors to help us gain more insight on to this dataset.

Collapsing Factors Together

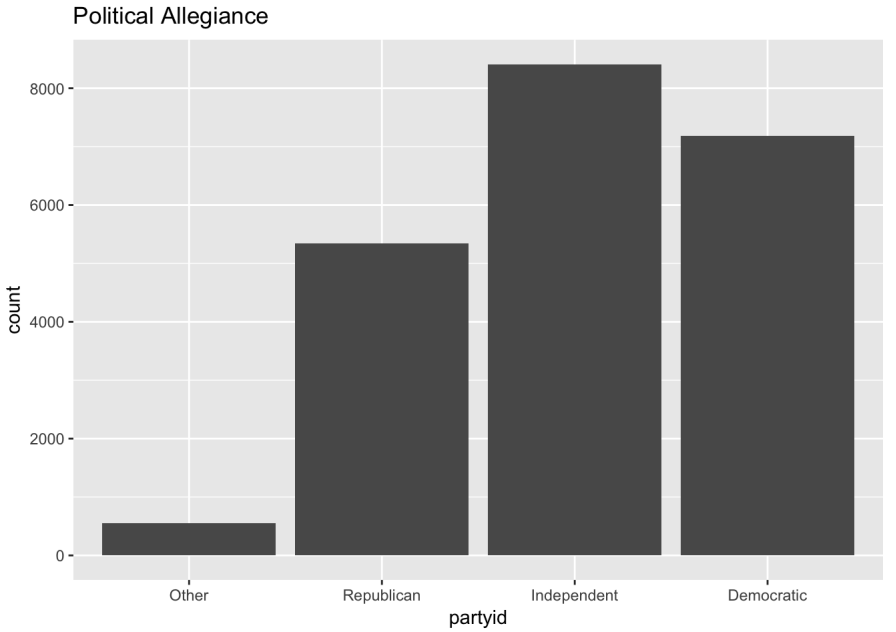
Let's look at a bar chart of the counts of the different partyid's the people have.

```
ggplot(gss_cat) + geom_bar(aes(partyid)) + ggtitle("Political Allegiance")
```



While this plot accurately gives us the data that we need by telling us how many people from the survey identify with which party and how strongly they identify with that party, that information may not be helpful for someone who, let's say, may be trying to predict the next winner of the presidential election. So, we can group some of these factors together to get a better understanding by using the fct_collapse function, which will group specified vectors together.

```
gssPoliticalParty <- gss_cat %>%
  mutate(partyid = fct_collapse(partyid,
    Independent = c("Ind,near rep", "Independent", "Ind,near dem"),
    Democratic = c("Not str democrat", "Strong democrat"),
    Republican = c("Strong republican", "Not str republican"),
    Other = c("Don't know", "No answer", "Other party")
    ))
ggplot(gssPoliticalParty) + geom_bar(aes(partyid)) + ggtitle("Political Allegiance")
```

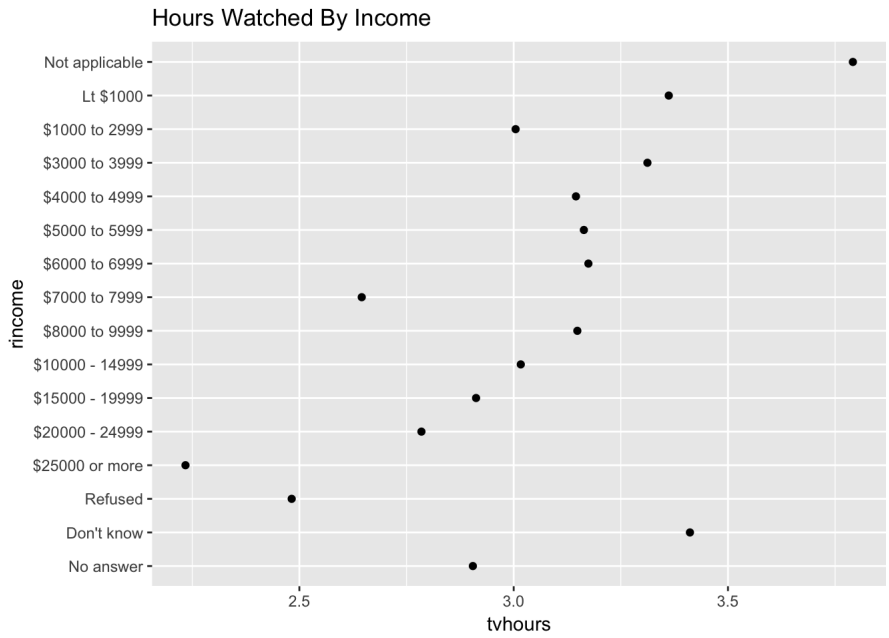


Now, we have a more useful chart that can tell someone where most people are right now, rather than a bunch of small groups.

Displaying the Factors Inorder

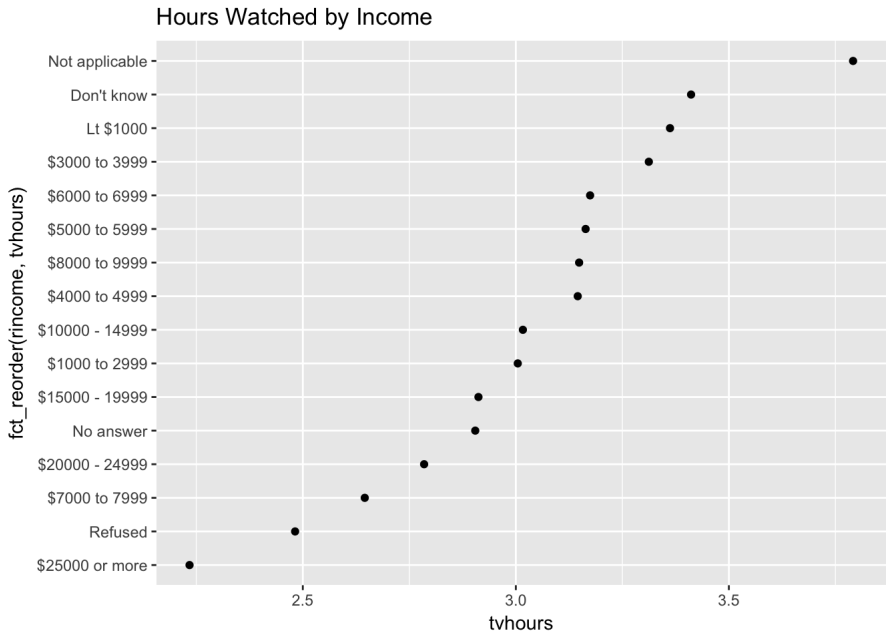
Let's answer another question that we may have with this dataset using factors. Let's say that we want to find out the average amount of hours of TV that people watch in each income bracket to see if there's a relationship between income and time spent watching TV. Let's just do it the standard way first.

```
rincome_hours <- gss_cat %>% group_by(rincome) %>% summarise(tvhours = mean(tvhours, na.rm = TRUE))
ggplot(rincome_hours) + geom_point(aes(tvhours, rincome)) + ggtitle("Hours Watched By Income")
```



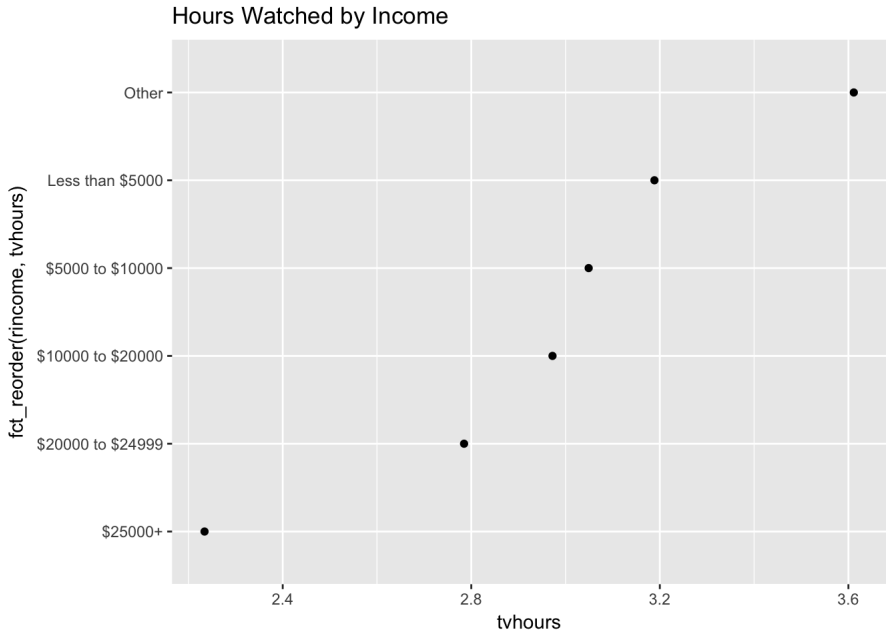
Doing this, it's very difficult to see what the relationship is between income and tvhours beacuse they aren't displayed in order of tvhours watched and there are so many factors that go into this. We can fix the first problem by using the fct_reorder function in our plot, which will arrange our plot by average hours spent watching TV.

```
ggplot(rincome_hours) + geom_point(aes(tvhours, fct_reorder(rincome, tvhours))) + ggtitle("Hours Watched by Income
")
```



However, this doesn't really tell us too much because of the fact that there are too many categories. So, we will use fct_collaps and this new function to make a more presentable plot and then we will perform the same procedure.

```
rincome_grouped_hours <- gss_cat %>% mutate(rincome = fct_collapse(rincome,
  "Less than $5000" = c("Lt $1000", "$1000 to 2999", "$3000 to 3999", "$4000 to 4999"),
  "$5000 to $10000" = c("$5000 to 5999", "$6000 to 6999", "$7000 to 7999", "$8000 to 9999"),
  "$10000 to $20000" = c("$10000 - 14999", "$15000 - 19999"),
  "$20000 to $24999" = c("$20000 - 24999"),
  "$25000+" = c("$25000 or more"),
  "Other" = c("Not applicable", "Don't know", "Refused", "No answer")
))
rincome_grouped_summary <- rincome_grouped_hours %>% group_by(rincome) %>% summarise(tvhours = mean(tvhours, na.rm
= TRUE))
ggplot(rincome_grouped_summary) + geom_point(aes(tvhours, fct_reorder(rincome, tvhours))) + ggtitle("Hours Watched
by Income")
```



Now, after combining the two functions we learned earlier, we have a plot that we can look at that is more understandable and it answers our question! There does appear to be a correlation between hours of tv watched and income brackets.

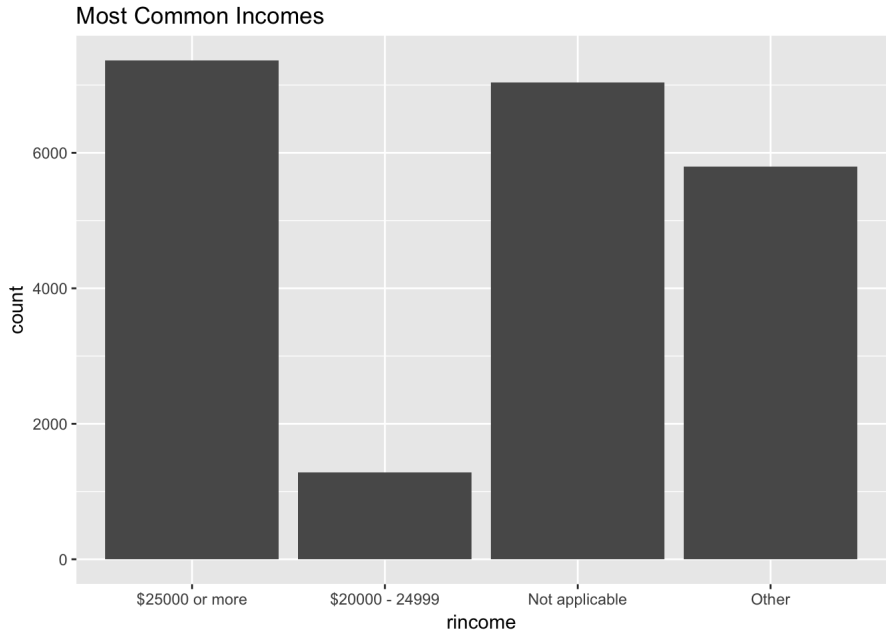
More Grouping

While fct_collapse is a very nice way of grouping our factors together, there are other ways of grouping.

Lumping Data Togehter

With fct_lump(), we can get the "n" most common frequencies of a factor, then group together the remaining pieces of data. For example, if we wanted to see the three most common income brackets, we can do this:

```
common_three_rincome <- gss_cat %>% mutate(rincome = fct_lump(rincome, n = 3))
ggplot(common_three_rincome) + geom_bar(aes(rincome)) + ggtitle("Most Common Incomes")
```



In this plot, we obtain the three most common incomes and the last bar, "Other" is just the sum of the counts of the remaining factors.

We can even get the least common incomes if we wanted to, by changing the value we give to "n" to negative.

```
gss_cat %>% mutate(rincome = fct_lump(rincome, n = -3)) %>% count(rincome)
```

```
## # A tibble: 4 x 2
##         rincome     n
##          <fctr> <int>
## 1     No answer   183
## 2 $7000 to 7999   188
## 3 $6000 to 6999   215
## 4         Other 20897
```

Cut Function

Finally, we have a way to "factor" numeric vectors with the cut function. For example, if we wanted to group people by age to see the average amount of tv watched, we can either "cut" people into equal frequencies or we can cut them in to equal intervals that we determine.

We will see how to cut them into equal intervals. Explore on your own to see how to get equal frequencies for each interval.

```
equal_age_intervals <- gss_cat %>% mutate(age = cut(age, 3))
table(equal_age_intervals$age)
```

```
##
## (17.9,41.7] (41.7,65.3] (65.3,89.1]
##        8961        8868        3578
```

As you can see, all intervals have equal length. Now we can go age group by age group to check the average amount of tv watched per day.

```
equal_age_intervals %>% group_by(age) %>% summarize(tvhours = mean(tvhours, na.rm = TRUE))
```

```
## # A tibble: 4 x 2
##           age  tvhours
##        <fctr>    <dbl>
## 1 (17.9,41.7] 2.659979
## 2 (41.7,65.3] 2.999352
## 3 (65.3,89.1] 3.727414
## 4          NA 2.631579
```

Conclusion

Factors are very powerful as they are much easier to work with than character vectors and we can manipulate them to our advantage in our data analysis. When we use factors, we can visualize data in a way that we weren't able to earlier.

Sources

https://www.youtube.com/watch?v=xkRBfy8_2MU

https://www.stat.berkeley.edu/classes/s133/factors.html

http://monashbioinformaticsplatform.github.io/2015-09-28-rbioinformatics-intro-r/01-supp-factors.html

http://r4ds.had.co.nz/factors.html

https://www.tutorialspoint.com/r/r_factors.htm

https://www.programiz.com/r-programming/factor

https://cran.r-project.org/web/packages/forcats/forcats.pdf