Stat 422/722: Recitation 2

Justin Khim

September 12th, 2017

1 Agenda

The R notes posted online by the instructor are fairly comprehensive. Here are the things we'll do today:

- make nice boxplots, both with the usual boxplot command and with ggplot;
- use the `mutate` function in `dplyr`, which I forgot to include in the previous set of notes; and
- learn a few additional tricks on linear regression in R.

But, before we get to these, we should discuss lists.

2 Lists

We briefly mentioned lists in R, but we didn't talk about them extensively. Lists are a very common data type in many coding languages. In R, people tend to define them explicitly less because vectors are preferable for storing numbers.

So, what's the difference between vectors and lists? Each element of a vector must be of the same data type, eg numeric, boolean, or string. We'll usually use numeric vectors (vectors of numbers), but we see some strings in categorical data and when creating names for plots. Further, having one type of data is nice because if we know that we have a numeric vector, then we know that functions like `mean`, `sd`, `hist`, and `boxplot` will work.

On the other hand, lists don't have such a restriction. This means that we wouldn't want to try to take the mean or make a histogram out of data in a list, because not every element in the list has to be a number. But, it might be nice in some cases to be able to store data of multiple types. For example, suppose that we want to store the attributes of height, weight, eye color, hair color, and favorite type of pizza for a person. We could use a list for this.

```
alice = list(65, 125, "blue", "blue", "pepperoni")
bob = list(70, 170, "brown", "black", "Hawaiian")
eve = list(64, 115, "brown", "brown", "pesto")
```

We can evaluate the ith element of a list using two brackets (on each side). For example, we can evaluate the first element of the lists.

```
alice[[1]]

## [1] 65

bob[[1]]

## [1] 70

eve[[1]]

## [1] 64
```

Of course, there's a special type of data in R for information like this: the data frame. Or really, the data frame is just a special type of list. Let's look at a subset of the house price data we had last time.

```
library(dplyr)

##
## Attaching package:  'dplyr'
## The following objects are masked from 'package:stats':
##
##    filter, lag
## The following objects are masked from 'package:base':
##
##    intersect, setdiff, setequal, union

kc.df = read.csv("kc_house_data.csv")
kc.df.smaller = kc.df %>%
                      select(id,
                             price,
                             bedrooms,
                             bathrooms)
```

Now, if this is a list, we should be able to access the second element using the brackets.

```
head(kc.df.smaller[[2]])

## [1]  221900  538000  180000  604000  510000 1230000
```

Recall that we only use the `head` function to limit the output to a reasonable number of entries (six). The output here were the first six house prices. You can play around with

other indices, but the point here is that a data frame is a list, and the ith element of that list is the ith column. This is worth taking a minute to think about: we can make lists where an element of the list is a vector.

```
example.list = list(c(1), c(2, 2), c(3, 3), "Hello, world!")
example.list

## [[1]]
## [1] 1
##
## [[2]]
## [1] 2 2
##
## [[3]]
## [1] 3 3
##
## [[4]]
## [1] "Hello, world!"
```

Now, you may be wondering why we need the double brackets. What if we just use single brackets?

```
example.list[3]

## [[1]]
## [1] 3 3
```

What this does is gives you a list that is one element long, and that list contains the ith element. Why might we want this to be the result? Well, we might want a subset of the list.

```
example.list[3 : 4]

## [[1]]
## [1] 3 3
##
## [[2]]
## [1] "Hello, world!"
```

It's clear here that we need the result to be a list because the elements of the list are of different types. The difference between single and double brackets is something to keep in mind, as it's an easy thing to mess up.

One final list thing (that's very specific to R) is naming. We can name the indices of our list to make them easier to access or interpret.

```r
attributes = c("height", "weight", "eyes", "hair", "pizza")
names(alice) = attributes
alice
```

```
## $height
## [1] 65
##
## $weight
## [1] 125
##
## $eyes
## [1] "blue"
##
## $hair
## [1] "blue"
##
## $pizza
## [1] "pepperoni"
```

And of course, we can access them with the usual bracket or dollar commands.

```r
alice[["pizza"]]
```

```
## [1] "pepperoni"
```

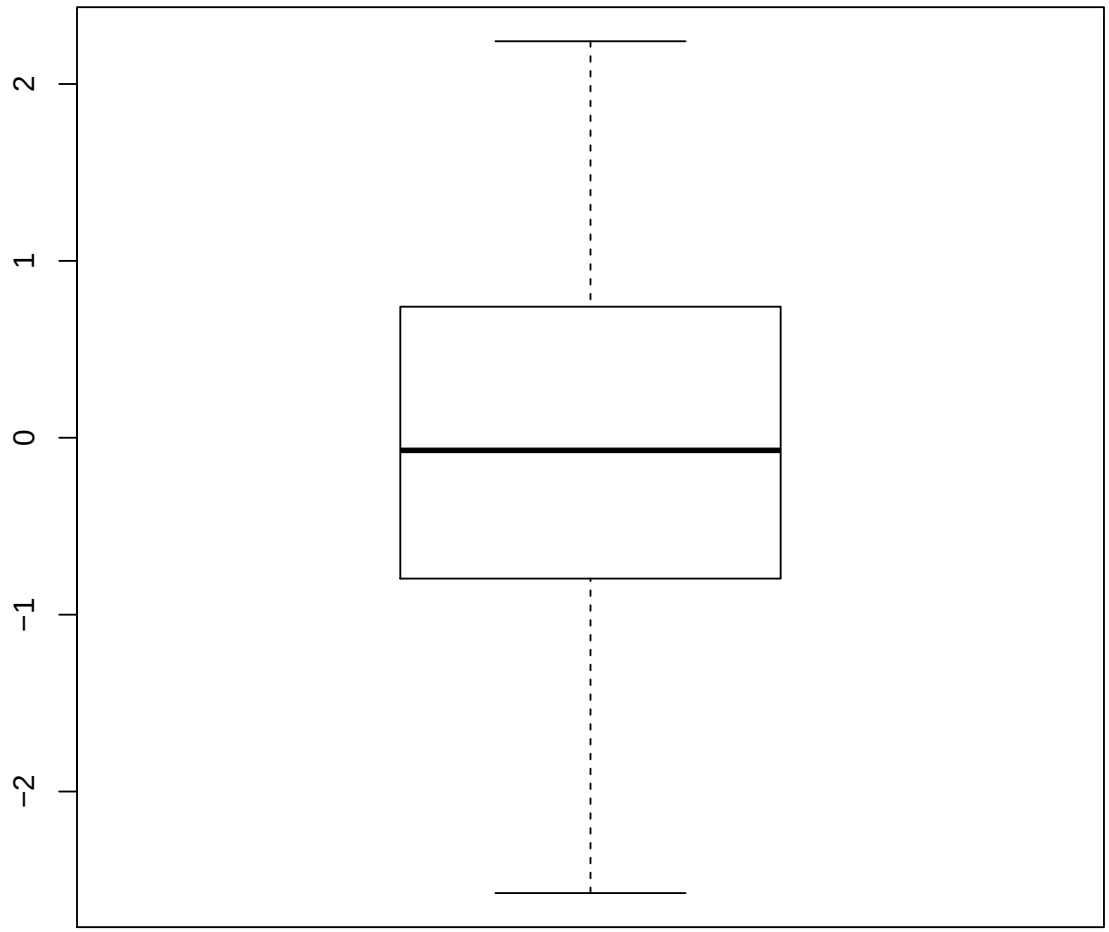
```r
alice$eyes
```

```
## [1] "blue"
```

3 Boxplots Revisited

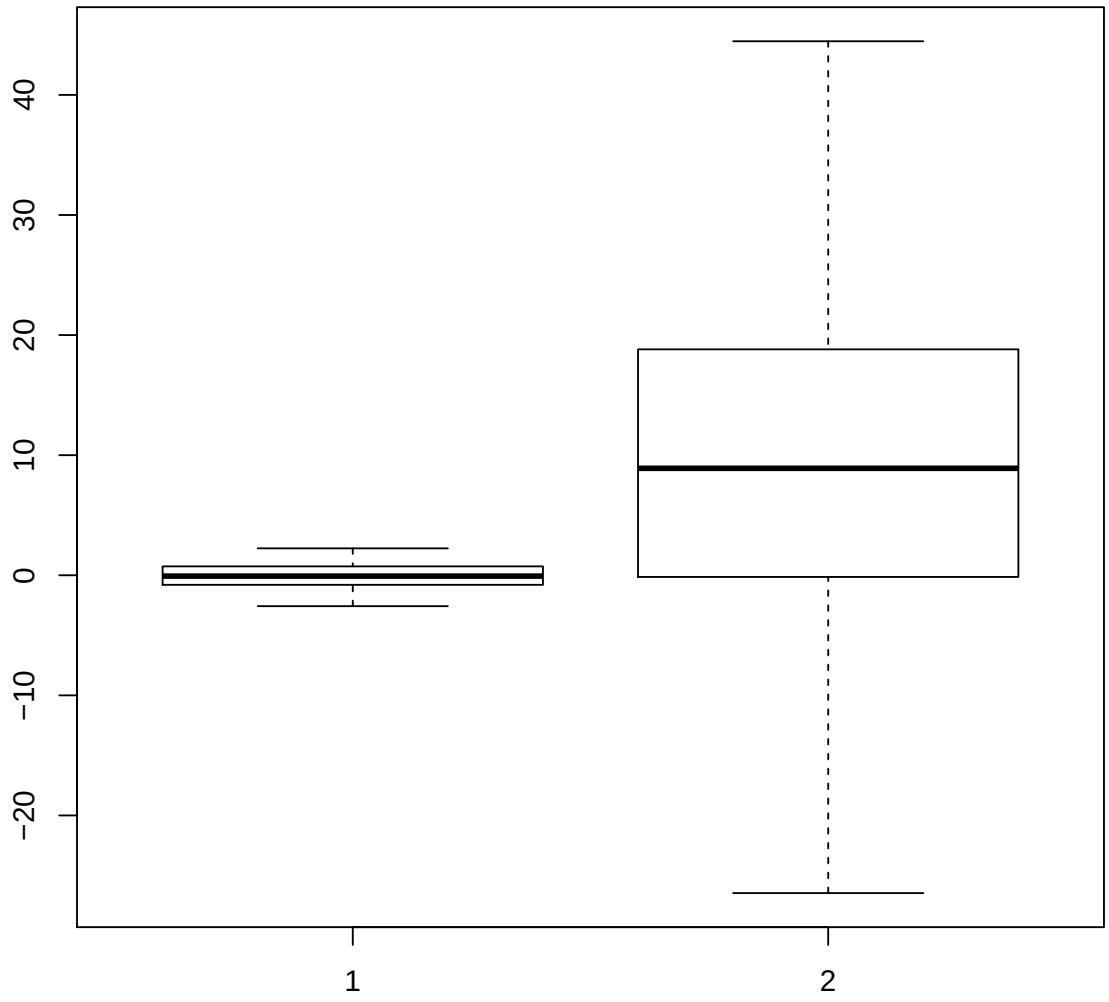
Now that we've done our meditations for the day on lists, we can revisit box plots. Let's look at a simple boxplot.

```r
x = rnorm(100, mean = 0, sd = 1) # vector of 100 standard normal rvs
y = rnorm(100, mean = 10, sd = 15) # vector of 100 normal rvs
boxplot(x)
```



So, this is how we make a boxplot to look at the distribution of one variable, the `x` in this case. Now, we can do it with two sets of data.

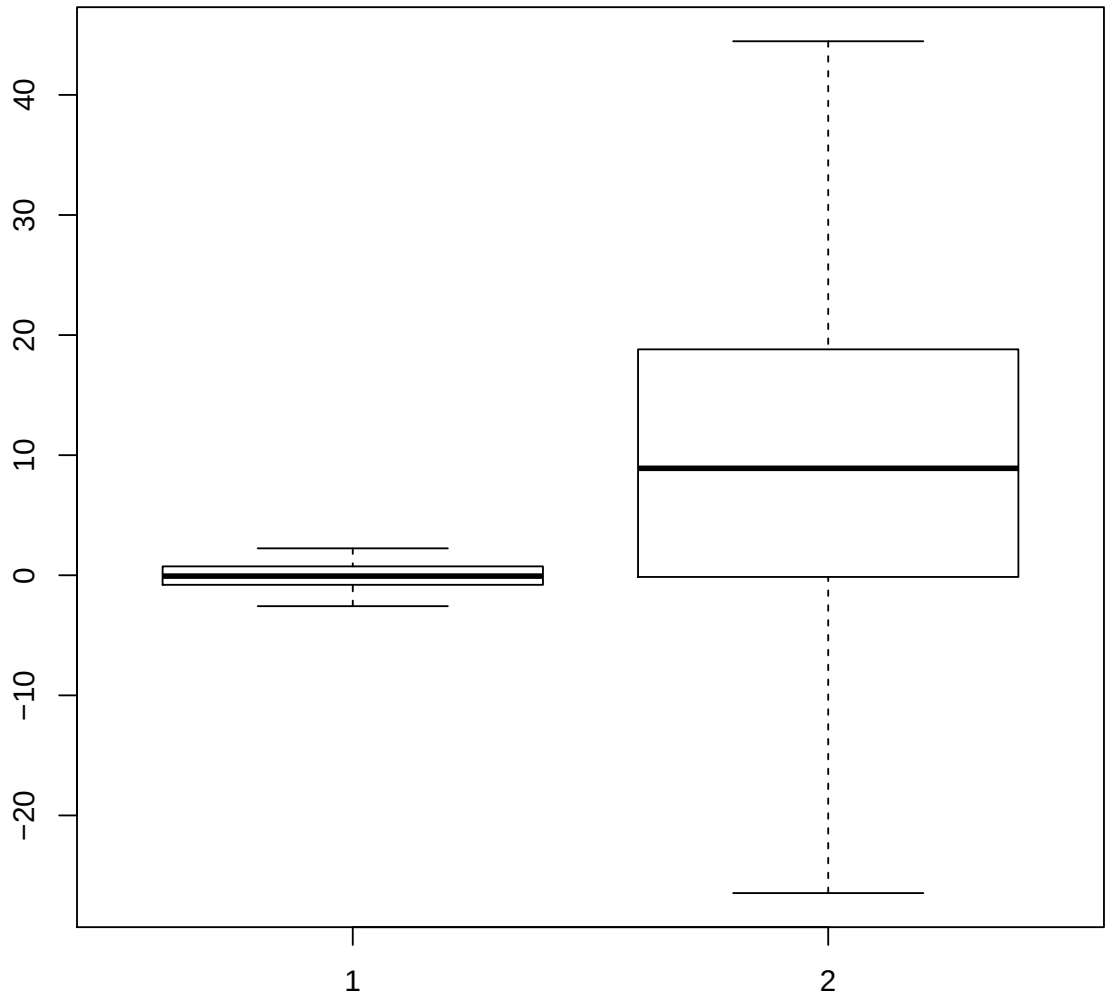
```
boxplot(x, y)
```



This gives two boxplots side-by-side. The first one corresponds to `x`, and the second, to `y`. We see that this makes sense, since the second boxplot has higher mean and standard deviation.

We can make this boxplot in another way using a list.

```
l = list(x, y)
boxplot(l)
```

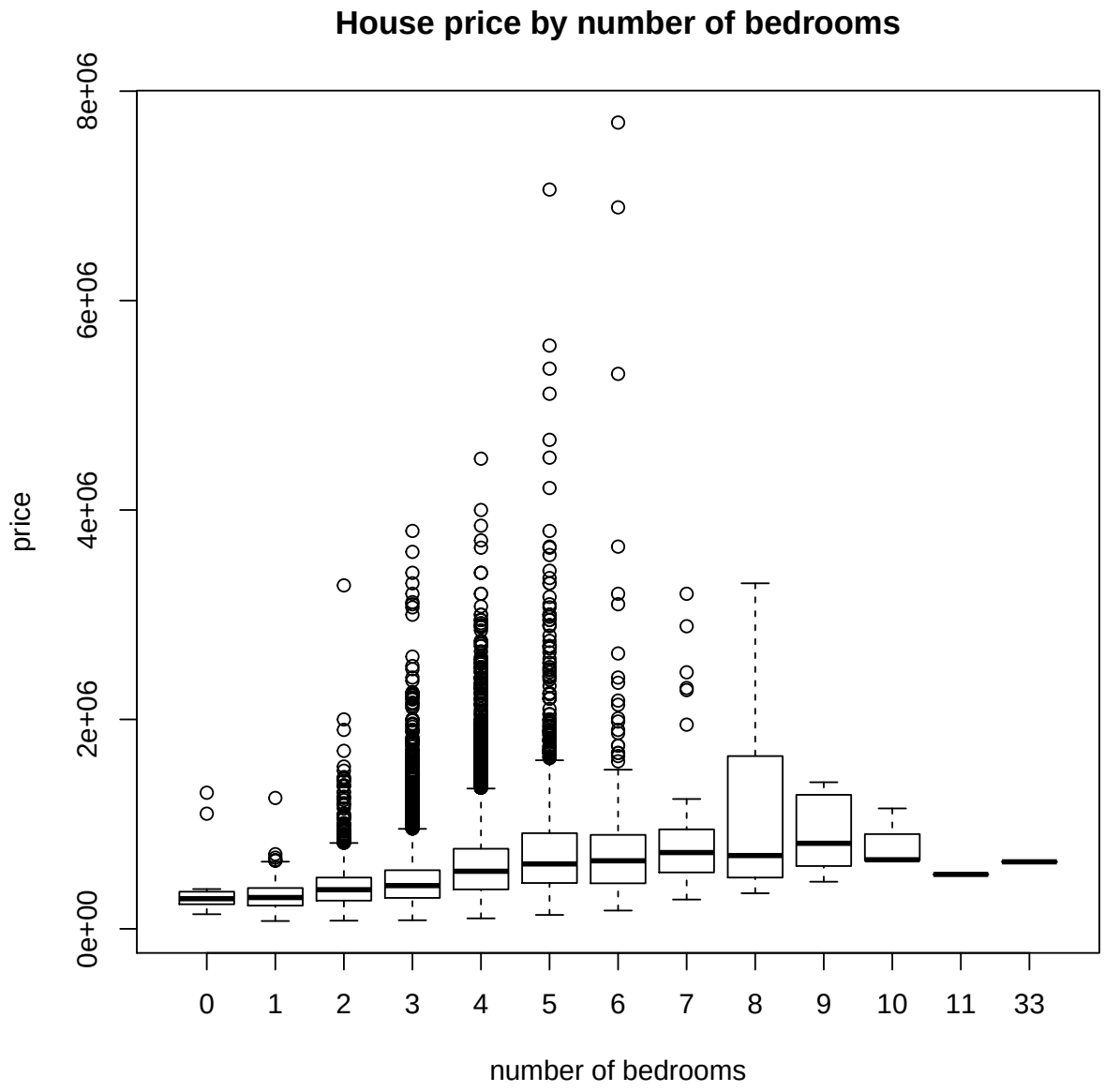


Here, `l` is a list containing `x` and `y` as elements. The `boxplot` function takes the list and plots each vector in its own boxplot.

So, now let's revisit the more complicated example from last time (it's slightly different here because we don't do all the other stuff) and break it down. For those of you who didn't like this example last time, I'll offer an alternative later.

```
price.by.room = kc.df %>%
                  group_by(bedrooms) %>%
                  summarize(price.list = list(price))
price.list = price.by.room$price.list
bedrooms   = price.by.room$bedrooms
boxplot(price.list,
```

```
        names = bedrooms,
        main = "House price by number of bedrooms",
        xlab = "number of bedrooms",
        ylab = "price")
```



This shows a nice plot with boxplots for the house price, broken down by the number of bedrooms. We should be able to deduce that `price.list` is the list that contains the vectors of prices. Here's the first element (the prices for the 0-bedrooms)

```
price.list[[1]]
```

```
##  [1] 1100000  380000  288000  228000 1300000  339950  240000  355000
##  [9]  235000  320000  139950  265000  142000
```

Now that's an expensive studio. So, there's no real mystery in the `boxplot` function. The only other things to note are that the optional argument `names` takes a vector of strings to use as the names of the individual boxplots.

So, we need to think about our `dplyr` functions. Remember that `group_by` groups the data frame into a tibble object (not that I know where that name comes from). In any case, the `summarize` function computes a function on each of the subgroups. Here, we asked for a list of the prices. For whatever reason, this creates a column `price.list` that is a list consisting of elements that are vectors of numerical data. Whew. If this doesn't make enough sense, we can try something that might make more or less sense.

We're going to try a new method to make these side-by-side boxplots, and to do this, we'll use a new package. So install `ggplot2`.

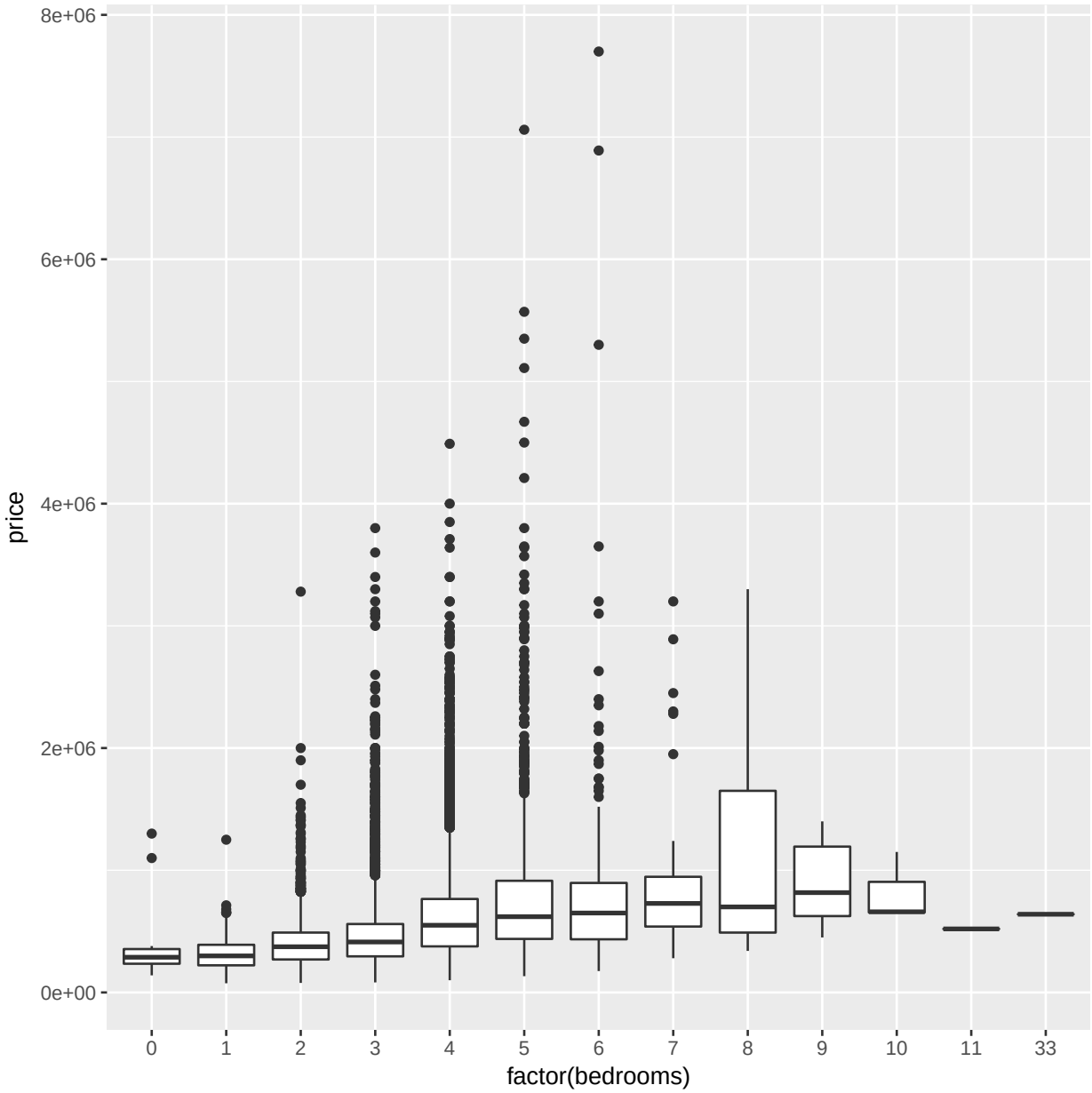
```
install.packages("ggplot2")
```

And then load it.

```
library(ggplot2)
```

Note that `ggplot2` is a part of the Tidyverse, which also includes `dplyr`. So, if you installed the Tidyverse, you'll already have `ggplot2` and will just need to load it. Anyways, we can make a plot by using the function `ggplot` and telling it what data frame it needs. To make a boxplot, we use the `geom_boxplot` function.

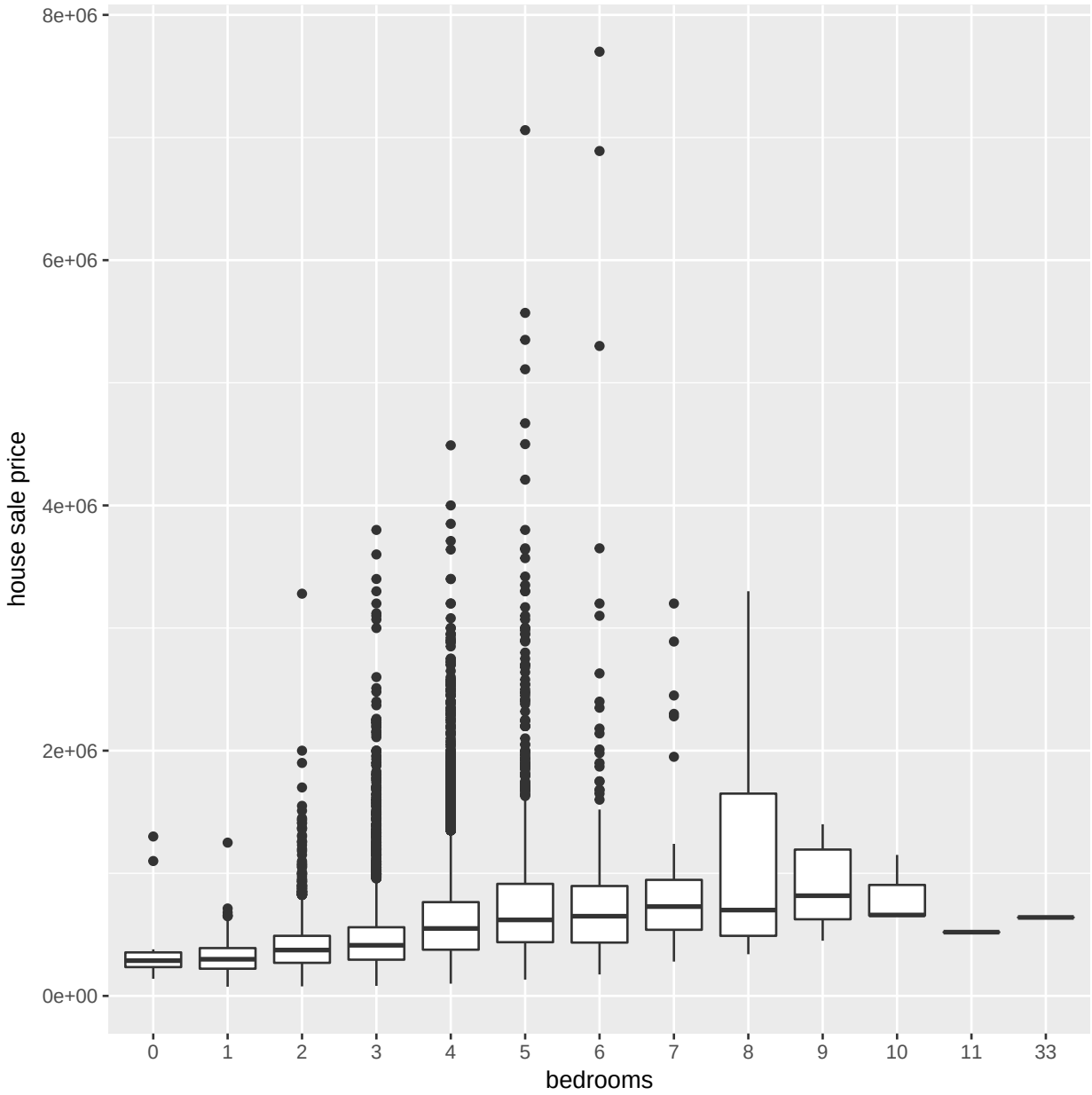
```
ggplot(data = kc.df) +
  geom_boxplot(aes(x = factor(bedrooms), y = price))
```



Note that here, the `aes` defines the aesthetic, which takes passes the data to `geom_boxplot`. Here, we're telling the boxplot to plot the bedrooms along the x-axis, treating each number as a categorical variable using the `factor` command. Next, the y-axis gives the prices, so we need to tell the boxplot that price is the function the data that we're interested in.

Now, let's clean up the boxplot.

```
ggplot(data = kc.df) +
  geom_boxplot(aes(x = factor(bedrooms), y = price)) +
  xlab("bedrooms") +   # set x-axis label
     ylab("house sale price") # set y-axis label
```



Here, we've simply modified the x-axis and y-axis labels using `xlab` and `ylab`. They each take a string and just make it the label. Simple–we didn't even need `dplyr`. Since this is the only instance (at the moment) where we need to plot something based on factors, we'll not provide the ways to make other plots using `ggplot`. However, they are generally considered to be nicer looking plots, and you may want to learn more about `ggplot` if you intend to display data in your work.

4 Using `mutate`

I mentioned `mutate` last time but neglected to tell you what it does. You can use `mutate` to add a new column to your data frame, or, more precisely, it outputs a data frame that

contains an additonal column. You can use this in case you have more sophisticated analyses, or if you need to use the `lm` function multiple times.

For a quick example, we can make an additional predictor that is the product of the square footage and the `waterfront` indicators.

```
kc.df.modified = kc.df %>% mutate(sqft_water_int = sqft_living * waterfront)
kc.df.modified$sqft_water_int[1 : 100]
```

```
##   [1]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
##  [15]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
##  [29]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
##  [43]    0    0    0    0    0    0    0 2753    0    0    0    0    0    0
##  [57]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
##  [71]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
##  [85]    0    0    0    0    0    0    0    0    0    0    0    0    0    0
##  [99]    0    0
```

So, we can see that one of the first 100 is nonzero. On the left side of the equals sign in `mutate`, you put the column name that you want to use for the new column. On the right hand side, you put in the equation for the new variable.

```
kc.df.modified2 = kc.df %>% mutate(bb.diff = bedrooms - bathrooms)
kc.df.modified2$bb.diff[1 : 100]
```

```
##   [1]  2.00  0.75  1.00  1.00  1.00 -0.50  0.75  1.50  2.00  0.50  0.50
##  [12]  1.00  2.00  1.25  3.00  1.00  1.00  3.00  1.00  2.00  2.25  0.25
##  [23]  2.50  0.50  0.75  1.00  1.25  2.00  1.25  1.50  0.50  0.50  2.25
##  [34]  2.00  1.25  0.50  3.00  2.00  3.00  1.50  1.50  1.75  2.25  2.00
##  [45]  2.00  0.50  2.00  1.50  1.25  0.50  2.00  2.50  1.25  0.25  2.75
##  [56]  1.50  1.50  0.50  1.75  1.00  0.75  1.25  1.25  1.25  0.75  0.25
##  [67]  1.50  1.50  2.00  2.75  1.75  1.00  2.00  1.50  1.25  0.00  2.25
##  [78]  2.00  2.00  1.50  0.50  1.50  1.00  1.50  2.00  1.50  2.75 -0.25
##  [89] -0.25  0.25  1.50  2.25  2.00  1.50  1.50  1.50  1.25  2.50  1.25
## [100]  0.50
```

Of course, don't put this last new column in a linear model with either the bedrooms or the bathrooms, since you'll just get the same model as including the bedrooms and bathrooms individually.

5 Regression Tips

The posted regression notes essentially tell you everything you need to know, but we'll point out a couple more features. First, let's fit a linear model.

```
kc.lm = lm(price ~ sqft_living + bedrooms, data = kc.df)
kc.lm

##
## Call:
## lm(formula = price ~ sqft_living + bedrooms, data = kc.df)
##
## Coefficients:
## (Intercept)  sqft_living     bedrooms
##     79282.0        314.2     -57112.9
```

The output here looks a bit weird. But, today's theme is lists, so we can guess this is a list as well.

```
names(kc.lm)

##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"

kc.lm[["coefficients"]]

## (Intercept) sqft_living    bedrooms
##  79281.9840    314.1588 -57112.8962
```

The index for the coefficients happens to contain a vector. Cool.

The other thing on linear models that I want to point out has to do with prediction on new data. The notes give an example of how to predict new price information based on a new data frame that is read in from a file, but you don't need to use a file to do this. We can just define a new data frame ourselves.

```
fake.sqft = c(4000, 2000)
fake.bedroom = c(4, 3)
fake.data = data.frame(sqft_living = fake.sqft, bedrooms = fake.bedroom)
predict(kc.lm, newdata = fake.data)

##         1         2
## 1107465.4  536260.8
```

Just note that for this to work, the new data needs to have the same column names (and types) as the ones used in the linear model of the original data. Note that it doesn't mean we need all the columns, but we do need the relevant ones appropriately named. Here's another example.

```
fake.data2 = data.frame(fake.sqft, fake.bedroom)
predict(kc.lm, newdata = fake.data2)

## Error in eval(predvars, data, env):  object 'sqft_living' not found
```

The lesson here is that the names of elements in lists really matter in R.