Codebooks, Measurement, Viz, Stats

Preamble

```
rm(list=ls())
setwd("C:/Users/19107/Desktop/R Stuff/2023 or Earlier/Teaching")
```

```
library(ggplot2)
set.seed(12345)
```

Agenda for today:

```
a) Reading Codebooks & Measurment
b) The Anatomy of a [gg]plot
c) Visualizing properties of Statistics
```

A) Reading Codebooks & Measurement

Choose one of the 4 codebooks listed to examine, choose 3 variables to get to know and be prepared to tell your classmates: the level of measurement, the concept it measures, a potential application for this variable. Also, tell us at least 1 thing about the dataset you learned from the codebook. Codebooks to choose from: CIRIghts, MAR, COWMID, ANES

B) The anatomy of a [gg]plot

There are 3 basic pieces to a ggplot (or any plot really)-

```
  i) The data layer

 ii) The Aesthetic layer

iii) The Polish layer
```

These pieces are connected by "+" symbols, which tells R to run them together

All ggplots begin by calling ggplot(), this can be left blank or you can supply a dataframe here which will tell ggplot we'll be pulling variables from this dataframe, it also means we can directly call our variables as opposed to using the $ symbol.

The aesthetic layer is where we specify the type of plot we want (geom_), I call this the aesthetic layer because geoms require aesthetics (aes) to run. The aesthetics are just arguments related to the variables you need to supply given the plots you are using e.g. you need to supply an x & y variable for a line but for a histogram you need only supply an "x" (you don't even need to use "x="). There are also options in this layer that can alter the appearance of your geom but only in limited set of ways which is why it is distinct from the polish layer.

The Polish layer tends to be the last set of layers you add to a plot, these are commands that allow you to customize your axis labels, the ticks on your axes, annotations, add or remove legends, title the plot, etc.

An example of plot refinement across commands

```r
# First things first I need some data!
vector = rnorm(100, 23, 1)
multiplier = runif(1, .01,2.3)
random_error= runif(100, 15,300)
vector = vector*multiplier
data=data.frame(x=vector,
                y= ifelse(vector>mean(vector),
                          ((2*sqrt(vector)-13)^2),
                          ((2*sqrt(vector)-13)^3)))
```

Data Layer

ggplot(data)

No need to run this because you can't plot with only a data layer.

Aesthetic Layer

```
ggplot(data)+geom_density(aes(x))
```

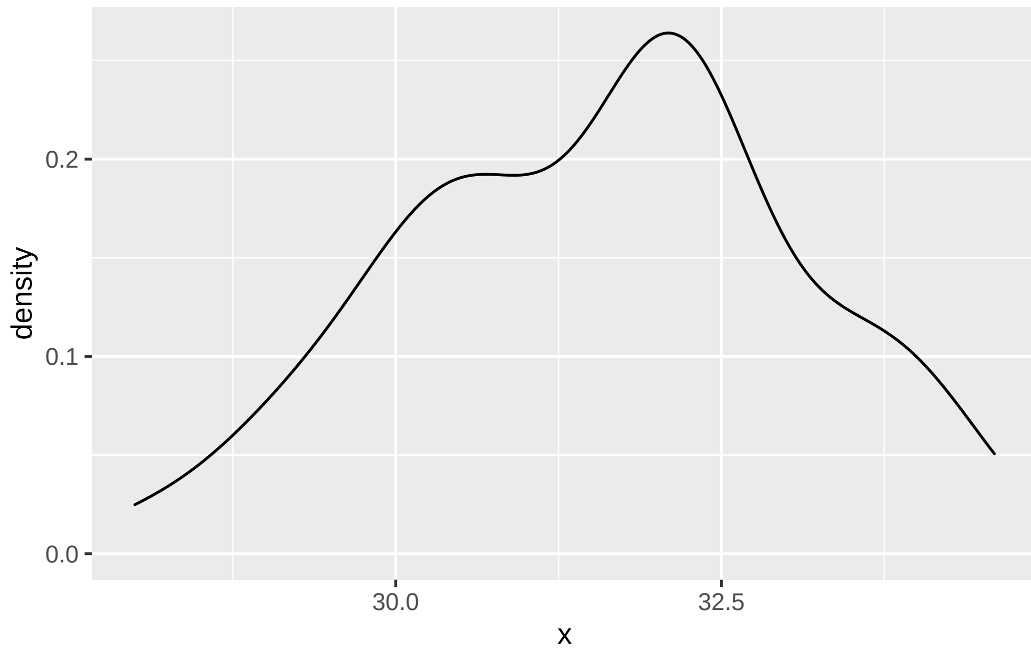


Figure 1: Just a basic density

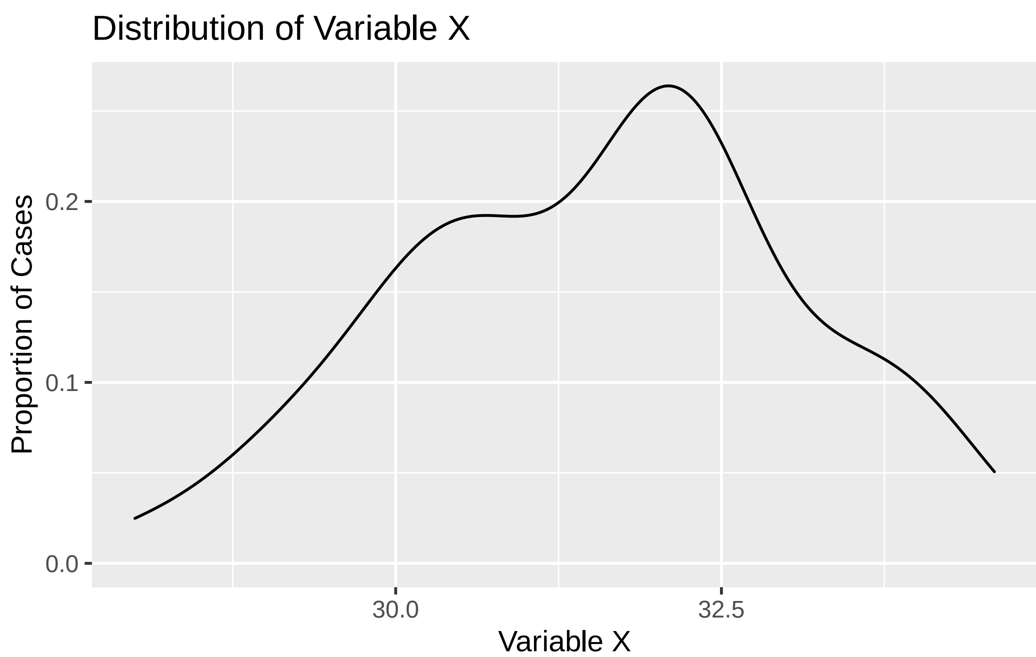
Typical aesthetic geoms:

- line
- histogram
- density
- point
- smooth*

*Smooth basically runs a regression on the x & y and returns the fitted line for that regression, uses some fancy stats but basically returns an approximation of the relationship between x & y and is smoothed as its name implies.

Polish Layer

```
ggplot(data)+geom_density(aes(x))+
  xlab("Variable X")+
  ylab("Proportion of Cases")+
  ggtitle("Distribution of Variable X")
```



Typical polish commands/geoms

- xlab
- ylab
- ggtitle
- annotate("text", label= "text to appear", x= coordinate for text, y= coordinate for text)
- geom_vline
- geom_hline

C) Visualizing properties of Statistics

Using a ggplot show me the following properties of certain statistics:

- Mode, the mode represents the highest point in distribution
- Range, the range is the interval between which all data in a distribution lies
- Standard Dev, the standard deviation governs the scale or spread of a distribution
- Coefficient of Variation, the coefficient of variation standardizes the relationship between distributions such that they are comparable and any two distributions with similar C.o.V.'s will be similar in appearance

My Answers:

In class I showed you my simple answers, but here I'll show my ideal answers

Mode

```
# Create some data with a known mode (I'm choosing 1 as my mode)

data=c(1,1,1,1,2,2,2,3,3,3,4,4,4,0,0,0,-1,-1,1)
# Table shows that 1 is in fact my modal category

table(data)
```

```
data
-1  0  1  2  3  4
 2  3  5  3  3  3
```

```
# I will represent this visually. I want to put a point on the mode, I
↪  want that point to be open rather colored in, and I would like an
↪  arrow pointing to it with some text that says "The mode of the
↪  Distribution", I'm new to arrows but there's always more code to
↪  learn, so I learned arrows just for yall.

ggplot()+geom_density(aes(data))+
  geom_point(aes(x=1, y=.209), shape=0)+
  geom_segment(aes(x=1,y=.2065, xend=1, yend=.15),arrow = arrow(length =
   ↪  unit(0.5, "cm"), ends="first"), linejoin ="mitre")+
  annotate("text", label="The Mode of the Distribution", x=1, y=.145)
```

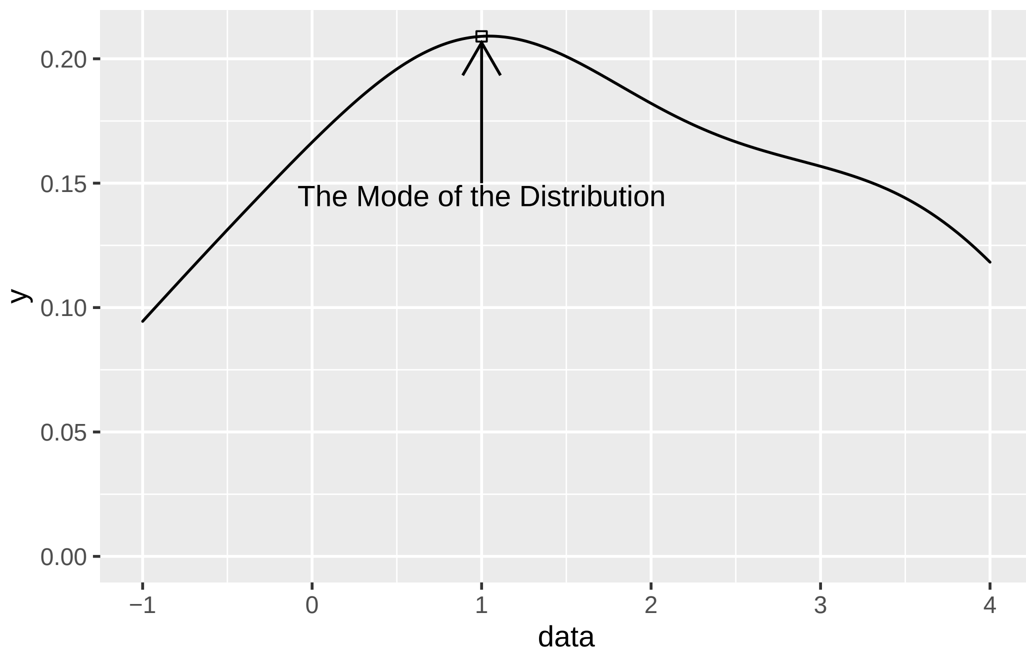


Figure 2: Density with mode indicated

Range

```
#I'll use the same data for this one as well. I want two vertical lines
↪  indicating the min and max i.e. the range and I want those lines to
↪  be dashed.

ggplot()+geom_density(aes(data))+
  geom_vline(aes(xintercept=range(data)), linetype=5)
```

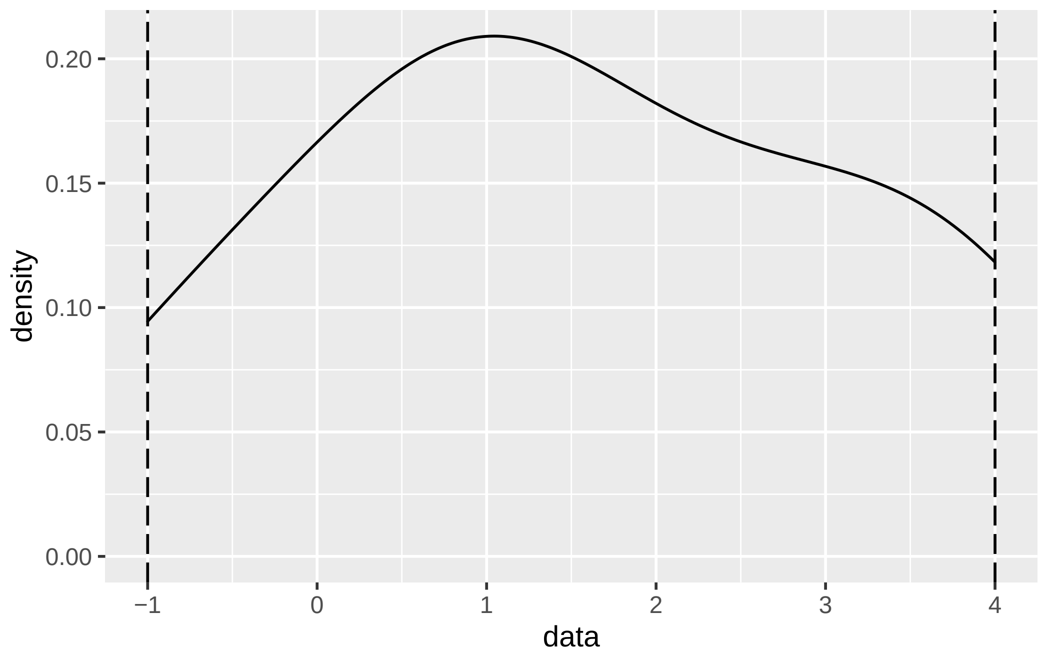


Figure 3: Density with range indicated

```
# alternatively I could use segments and make a little square
 ↪  bracket-like object.

ggplot()+geom_density(aes(data))+
  geom_segment(aes(x=range(data), y=c(.095,.12), xend=range(data),
   ↪  yend=c(.05,.05)), linetype=5)+
  geom_segment(aes(x=-1,y=.05,yend=.05,xend=4), linetype=5)+
  annotate("text", label="The Full Range of Values in this
   ↪  Distribution", x=median(data)+.5, y=.04)
```

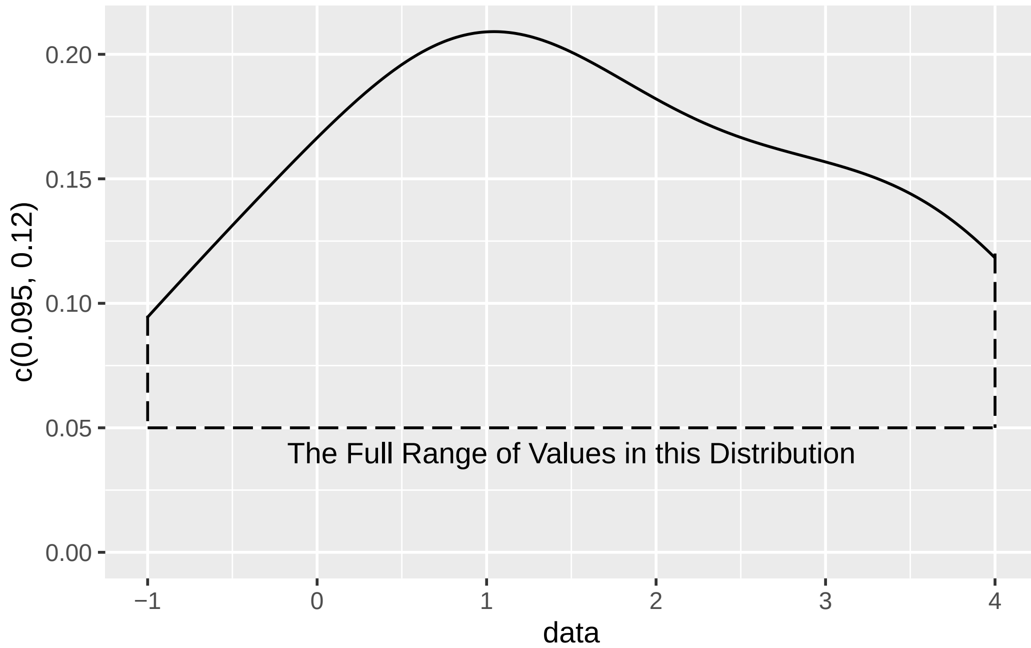


Figure 4: Alternative density with range indicated

Standard Deviation

```
# Here things get a little complex, the lazy way to do this is what I
↪  showed you in class, you use the rnorm command to make 2 random
↪  normal distributions with the same mean but different standard
↪  deviations. I will do this the slightly harder way using my own
↪  generated vectors. I'll modify the one vector I've been using and
↪  then create another
data=c(10,1,1,1,1,2,2,2,3,3,3,4,4,4,0,0,0,-1,-1,1)

length(data)
```

```
[1] 20
```

```
mean(data)
```

```
[1] 2
```

```
sd(data)
```

```
[1] 2.44949
```

```
# I was still lazy...but it does what I need
data2=c(rep(1.6,17),1.0,2.2,10)
length(data2)
```

```
[1] 20
```

```
mean(data2)
```

```
[1] 2.02
```

```
sd(data2)
```

```
[1] 1.888358
```

```
# all other stats are the same except standard deviation

# I want two arrows and some text to make things look nice my desired
 ↪  text is too long, but a nice person on stack overflow gave me this
 ↪  function called "wrapper" which will wordwrap my text
#
 ↪  https://stackoverflow.com/questions/25106508/ggplot2-is-there-an-easy-way-to-wrap-anno

wrapper <- function(x, ...) paste(strwrap(x, ...), collapse = "\n")

ggplot()+geom_density(aes(data))+
  geom_density(aes(data2), linetype=5)+
  geom_segment(aes(x=5,y=.25, xend=2.6, yend=.25),arrow = arrow(length =
   ↪  unit(0.5, "cm"), ends="last"), linejoin ="mitre")+
  geom_segment(aes(x=5,y=.25, xend=3.5, yend=.15),arrow = arrow(length =
   ↪  unit(0.5, "cm"), ends="last"), linejoin ="mitre")+
```

```
annotate("text", label=wrapper("Standard Deviation Moderates the
↪  'Spread' of Distributions", width=6), x=5.91,y=.27)
```

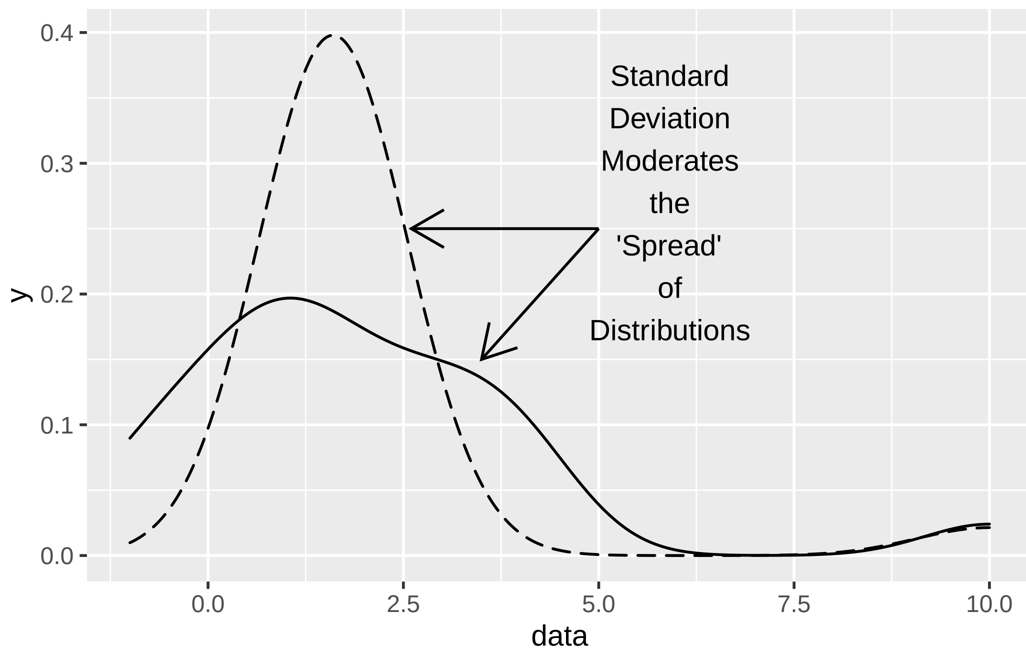


Figure 5: Pair of densities with different standard deviations

Coefficient of Variation

```
# This one will allow me to be lazy, I will use data2 and the patchwork
↪  library, patchwork allows you show plots side-by-side or ontop of
↪  eachother basically you can plots together using simple operators,
↪  for side-by-side you use | e.g. ggplot_object1|ggplot_object2, for
↪  stacking them you use / . the other secret here is that I can
↪  guarantee the same coefficient of variation by making one vector a
↪  multiplicative transformation of the other.
library(patchwork)
sd(data2)/mean(data2)
```

```
[1] 0.9348305
```

```
data3=data2*2

sd(data3)/mean(data3)
```

```
[1] 0.9348305
```

```
# these vectors have different standard deviations, different means, and
↪  different
# ranges, but the ratio of their standard deviations to their respective
↪  means is
# the same, therefore when plotted they will look the same.
ggplot()+geom_density(aes(data2))|ggplot()+geom_density(aes(data3))
```

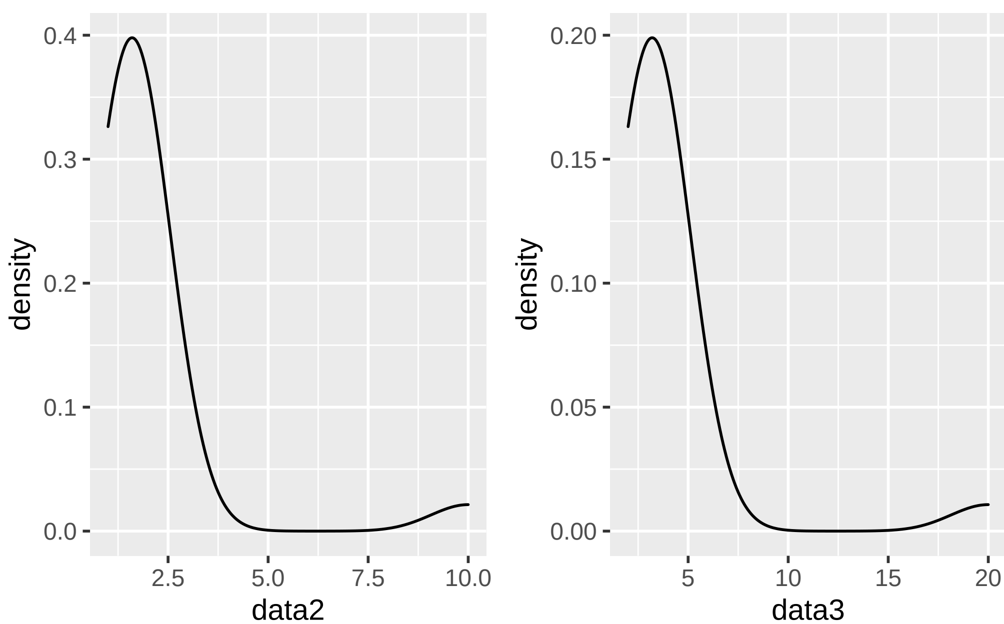


Figure 6: Pair of densities with matching coefficients of variation

```
# Here's another example where I haven't been so brazen with my
↪  manipulation
# These vectors are structured similarly but its not immediately obvious
# what their relationship is. Yet they have the same ratio of standard
```

```
# deviation to mean and so they're distributions will appear incredibly
# similar.

data2=c(-1,-2,100,100,0.1,1,2)
data3=c(-50,-25,1405,1400,-1,25,50)
ggplot()+geom_density(aes(data2))+xlab("Data2
↪  Values")|ggplot()+geom_density(aes(data3))+xlab("Data3 Values")
```

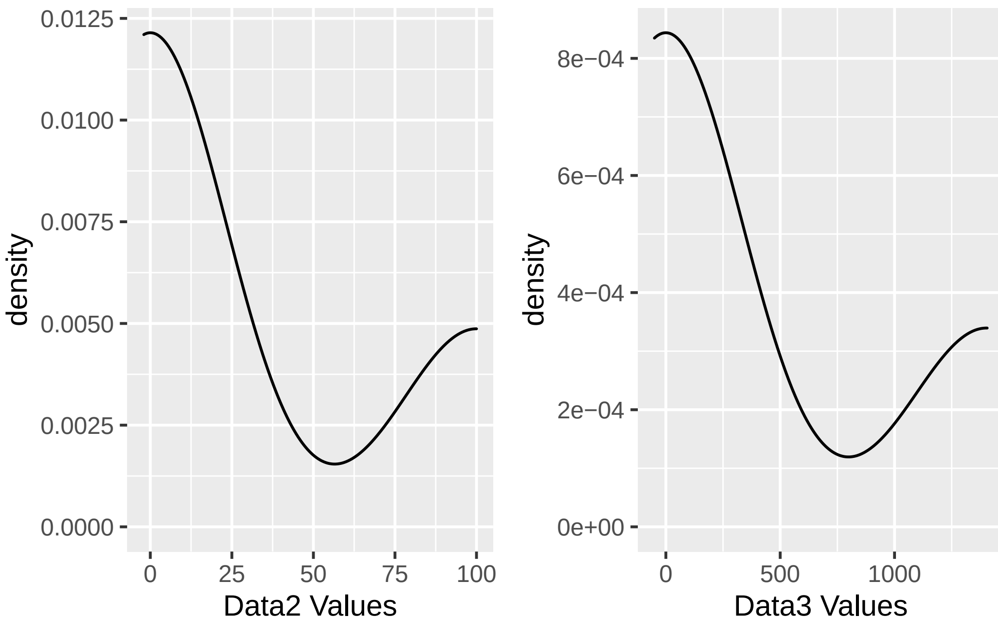


Figure 7: Pair of densities with matching coefficients of variation

Advanced Code Section

Don't hurt your brain too much here

How to simulate that deviations from the mean are minimum

Step One: Create some data

```
set.seed(123456)

# When using random number generation you should "set" your "seed" don't
↪ worry too much about this for now. Basically it just ensures
↪ replicability.

#Now I'll create the data, and I'm making my data complex for two
↪ reasons:

# 1) Because I can, and
# 2) To 'randomize' the data's parameters. When you generate random data
↪ you often specify its exact parameters and I wanted to introduce
↪ some extra variation, in particular, I wanted to make it so that the
↪ mean I specify originally for my random data is not the mean of the
↪ data I test.

# I chose a random distribution for my data, gamma, mainly because I
↪ don't encounter gamma often, generally because of the things I
↪ study.

vector=rgamma(100,10,1)
print(vector)
```

```
 [1] 12.243516  8.668214  7.067265 17.710082 12.246060 13.975745 18.779480
 [8] 13.441923  8.231874  5.673137  9.329000 13.426516  6.745112  7.369518
[15]  5.667264 11.303387  7.318213  7.907405 10.093182 10.979312  8.228438
[22] 13.158537 15.873850 12.769068 15.386546 12.103538  3.966215 14.916928
[29] 12.886314 12.990288  6.371860  2.489493 10.034471  6.938387  6.692368
[36]  6.756078  4.856293 14.550790  6.982618 14.535441 14.828879  8.482206
[43]  8.745563  8.736953 15.891522  7.175681  9.882298  7.095760  7.808778
[50]  8.939351  8.947445  7.361746 10.829281 12.329898  9.279990  5.577450
[57] 10.608223 11.775915  7.465305  3.728853  7.711956  5.268383 13.722531
```

```
[64]  6.032889 12.293654 10.880405 11.179726  8.733585 10.347973 10.956054
[71] 11.771949  8.084734 12.349975 10.672180 15.007916  9.686985  9.900855
[78]  7.058652  7.371297 11.378061 12.348561 12.771568  5.985738 10.367252
[85]  7.956276  7.016690  9.426243  8.041562  4.220404  6.812404  7.848112
[92]  8.950051  7.644008 15.181425 14.676794 16.550777  7.473559  6.725767
[99] 10.024161 11.298896
```

```r
# I also create another vector here which is distributed uniformly with
 ↪ a min of 0 and a max of 10, the thing about uniform distributions is
 ↪ that as their name implies they are uniform, i.e. every value in the
 ↪ range has an equal probability of occurring this gives it many
 ↪ useful properties. I'm using it as a multiplier to scramble up my
 ↪ original vector.

a=runif(100,0,10)

# I make vector equal to itself multiplied the "a" vector, this means
 ↪ that some values will increase by a factor up to 10 others will
 ↪ decrease (perhaps even to 0).

vector=vector*a
print(vector)
```

```
 [1]  97.8656875  68.1525913  62.2508917 122.5598171  63.6895784   8.3904964
 [7]  30.9837886 109.0834029  28.1406686  23.6969842  49.9254265 129.7642723
[13]  57.5044620  42.0320172  22.4284303  23.7560298  50.5253941  77.0409349
[19]  94.0058163  60.1921585  27.2330420  47.5436175  47.5906864 109.0221604
[25]  11.0466166 106.2216582  38.2778522  31.6527791  65.8981245 101.4495586
[31]  52.2973476   0.5731070  55.2285827  25.1834602  29.6949021  43.8029109
[37]  14.9461540  87.5781583   4.6013575 121.0228006  45.4170384  26.4318158
[43]  14.7900554  46.5271563  93.9443441  31.2675442  90.2971451  63.7524609
[49]   6.3062724  21.0682267  74.1740905  69.0787907  71.6978106  99.0526962
[55]  13.9802578  43.9971021  59.0229585  43.0929425  57.8502584  20.1353075
[61]  68.8912661  29.5241202 130.0064384   0.4240110  63.9693135  39.8401850
[67]  63.6485153   8.2321058 101.8078583   3.0746254  96.2775457  58.1196480
[73]   6.0051880   4.6057203  87.7724444  33.4370830  49.1896555   4.0373299
[79]  68.7467366 110.0789749  72.7002452 106.6947209  25.3255742  38.4235414
[85]  13.1488895  62.0025471  40.5563727  29.7032171   0.6222768  38.5627202
[91]  30.9145429  29.8492432  22.9364545 148.0717675  38.0648399 117.8489626
[97]  51.7306252  17.5271605  36.0482427  54.3778302
```

Step Two: Simulation

```
# To demonstrate the property of mean I want to, I'll need to run a
 ↪  'simulation' where I rotate through a bunch of values using them to
 ↪  generate deviations which I will square and sum. To do this I will
 ↪  use the "for" command, don't think too hard on this for now but for
 ↪  commands are extremely powerful and useful. This is called
 ↪  "looping", be careful if you choose to play with this code as it's
 ↪  not hard to create an 'infinite loop' which will make your R very
 ↪  unhappy.

# These three objects are to help me run my loop, result & k are where
 ↪  I'll store information from my loop, the counter helps me index
 ↪  (which we covered earlier in the semester).

result=NULL
k=NULL
counter=1

# This is my for-loop, basically I tell it create a temporary object
 ↪  named i, assign it each value in this seq() command (1 at a time),
 ↪  and then do all the stuff between these curly brackets.


for(i in seq(from=mean(vector)-100,to=mean(vector)+100,by=1)){
  result[counter]=sum((vector-i)^2)
  k[counter]=i
  counter=counter+1
}
```

Step Three: Plot

```
#The real useful thing in this code and my justification for showing it
 ↪  to you at this stage in your development is that you'll notice I
 ↪  don't use fixed numbers for almost anything I'm doing here, instead
 ↪  I have set this code up to work with whatever vector is assigned to,
 ↪  you can test it yourself by changing the values of the "vector"
 ↪  object yourself, no matter what you put in this code will work, it
 ↪  is self-contained. This is possible because I use nested functions
 ↪  rather than explicit numbers, this will make more sense when you
 ↪  look at the plot.

#NOTE: Every time you run the loop you should run the lines above that
 ↪  reset my helper objects.

ggplot()+geom_line(aes(x=k,y=result))+
  geom_point(aes(x=k[which(result==min(result))],y=min(result)),
   ↪  shape=0)+
  geom_vline(aes(xintercept=mean(vector)), linetype=5)+
  annotate("text", x=mean(vector)-.25*sd(vector), y=mean(result),
   ↪  label=paste0("mean of vector, ", round(mean(vector),3)),
   ↪  angle=90)+
  xlab("Any constant, k, to substract from a random vector")+
  ylab("The sum of squared deviations from k")
```

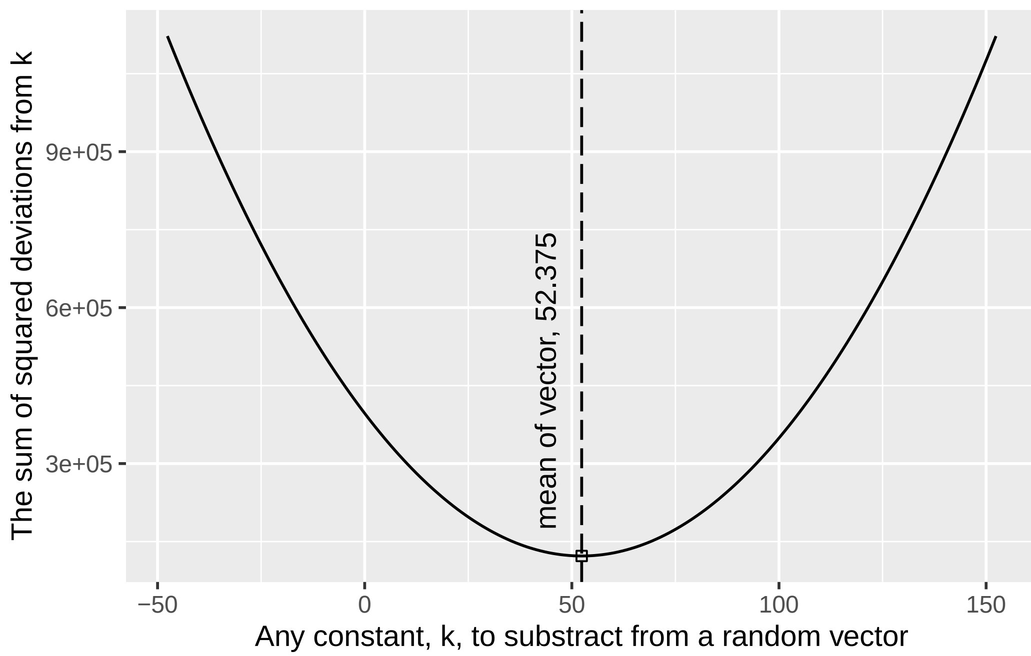


Figure 8: Plot showing the minimizing properties of the mean

```
# The plot is "programmed" to place a square point over the minimum sum
↪  of squared deviations from the vector i.e. the minimum of the
↪  "result" object this is achieved using min(result) as the y and a
↪  combination of which() & indexing for the x. Likewise the plot will
↪  always place a dashed vertical line at the mean of the vector object
↪  (which will coincide with the point command above due to the
↪  properties of the mean). This "programming" also applies to the
↪  annotations, I set the location of the annotations to be based on
↪  the mean & stand dev. of the vector and result objects.
```

One of the key takeaways from this section is that although the APPLICATION and EXECUTION of the code is quite advanced, the tools I used were quite elementary and in fact most of them are ones I've already taught you either directly or incidentally.