Distribution of Hourly Earnings by Age

Econ 440 - Introduction to Econometrics

Patrick Toche

19 February 2022

Objectives

1. Compute the marginal distribution of Age.
2. Compute the mean of AHE for each value of Age.
3. Plot the mean of AHE versus Age.
4. Compute the variance of AHE.
5. Compute the covariance between AHE and Age.
6. Compute the correlation between AHE and Age.

Why use R

The convenience of programming over spreadsheets is that you can run multiple calculations in a few lines of code and, as a result, you can reproduce your results quickly, and share your calculations easily. With a spreadsheet, you typically need to use the menus and mouse multiple times and replication is not so straightforward.

`R` is open-source, free software that is popular for statistical calculations. An alternative is `Python`, also open source and free.

Each have their strengths and weaknesses.

One of the best things about `R` is the `RStudio` interface and the `tidyverse` family of packages, particularly the plotting library `ggplot2`.

One of the best things about `Python` is its huge and increasing following, particularly in machine learning.

For data-heavy project, consider `Julia`, another wonderful open-source project.

Set Up Working Directory

The working directory is where R/Rstudio will search for data files and where it will save objects you create.

If you're on Windows, set your working directly with:

```
    setwd("c:/R/workspace")
```

If you're on MacOS, use

```
    setwd("~/R/workspace")
```

where ~ is a short-hand for the path to the user directory.

The condition below checks your operating system and sets a working directory accordingly.

```
# useful if you switch from Windows to Mac/Linux:
if(.Platform$OS.type == "windows"){
    setwd("c:/R/workspace")
} else {
    setwd("~/R/workspace")
}
```

Check that you're in the right place with

```
    getwd()
```

Set Options

To reduce the number of significant digits that are printed in the console, you may set the `digits` option:

```
library("readxl")
options(digits=4)
```

If you are working with time series data, you may find the `digits.secs` option useful,

```
library("readxl")
options(digits.secs=2)
```

Another useful option to limit or increase the output printed to the console is `max.print`,

```
library("readxl")
options(max.print=5)
```

Note that in R, the period is a character like any other. In Python it is not, so the naming convention would most likely be one of `maxprint` or `MaxPrint`. In Python, the use of underscores, as in `max_print`, is usually preferred for functions, with the "camel' ' notation preferred for variable names, but not everyone follows these guidelines. In R, the use of dots in variable names and function arguments is popular.

Install Packages

In R, packages are also called libraries.

In order to use a package, you first need to install it:

```
add.packages("readxl")
```

And before you use it, you must load it with the `library()` function,

```
library("readxl")
```

Once in a while, remember to update your existing packages, with

```
update.packages()
```

The above will attempt to update all your packages. This is generally a good idea, though it could occasionally break your code if the package has experienced an overhaul.

If you need to install a package from source – an older version of a popular package or an experimental package not available via the usual distribution channels:

```
url <- "http://cran.r-project.org/src/contrib/Archive/ggplot2/ggplot2_0.9.1.tar.gz"
install.packages(url, repos=NULL, type="source")
```

Not recommended unless an update has broken your code and you are under a tight deadline!

Read the Data

The data is in `xlsx` format: One way to read it is with the `readxl` library. This creates a `tibble` (part of `tidyverse`) rather than the standard `dataframe` (base R).

```
library("readxl")  # remember to load the package beforehand
data <- read_xlsx("Age_HourlyEarnings.xlsx")
```

- We are using the `read_xlsx` function from the package `readxl`.
- We are assigning the result of applying the `read_xlsx` function to the variable name `data`

In R, assignments are bi-directional, so we could also do:

```
library("readxl")
read_xlsx("Age_HourlyEarnings.xlsx") -> data
```

An assignment could also be done with the equal sign,

```
library("readxl")
data = read_xlsx("Age_HourlyEarnings.xlsx")
```

Was the data loaded correctly?

```
View(data)  # Note the capital V in View!
```

Read the Data

More control: takes some trial and error to get it right!

```
library("readxl")
df1 <- read_xlsx("Age_HourlyEarnings.xlsx",
                 col_names=TRUE,
                 skip=1,
                 trim_ws=TRUE)
```

- Set `col_names=TRUE` to use the ages as column names
- Skip the first line with `skip=1`
- Trim white space with `trim_ws`, because white spaces can mess things up in ways that are difficult to debug.
- See more options with `?read_xlsx`

Quick View of the Data

Our data may be accessed with the name `df1`. We will clean it and, as we do, overwrite it at each iteration. However, when you are debugging your own data, it is recommended that you use a different name, say `df1.tmp`, and check that everything is as expected before renaming it `df1` and overwriting the data.

The dataframe `df1` is in **wide format**. We will convert it to **long format** and name the new dataframe `df2`. We will use `df1` or `df2` whenever the format is most convenient for our purpose.

Quickly view a slice of the data with `head(df1)`:

```
head(df1)
#> # A tibble: 6 x 13
#>     AHE    `25`    `26`    `27`    `28`    `29`    `30`     `31`    `32`    `33`
#>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>    <dbl>   <dbl>   <dbl>
#> 1     5 0.00298 0.00218 0.00204 0.00211 0.00145 0.00153 0.00131  0.00102 0.00145
#> 2     6 0.00116 0.00167 0.00109 0.00109 0.00131 0.00109 0.000509 0.00174 0.00102
#> 3     7 0.00247 0.00262 0.00254 0.00167 0.00283 0.00240 0.00174  0.00189 0.00204
#> 4     8 0.00240 0.00225 0.00167 0.00240 0.00138 0.00276 0.00218  0.00174 0.00211
#> 5     9 0.00356 0.00327 0.00283 0.00291 0.00320 0.00240 0.00262  0.00196 0.00305
#> 6    10 0.00516 0.00523 0.00400 0.00523 0.00443 0.00451 0.00458  0.00414 0.00480
#> # ... with 3 more variables: `34` <dbl>, ...12 <lgl>, ...13 <lgl>
```

Quick View of the Data

View less or more of the data by setting the second argument of `head()`

```
head(df1, 2)
#> # A tibble: 2 x 13
#>     AHE    `25`    `26`    `27`    `28`    `29`    `30`     `31`    `32`    `33`
#>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>    <dbl>   <dbl>   <dbl>
#> 1     5 0.00298 0.00218 0.00204 0.00211 0.00145 0.00153 0.00131  0.00102 0.00145
#> 2     6 0.00116 0.00167 0.00109 0.00109 0.00131 0.00109 0.000509 0.00174 0.00102
#> # ... with 3 more variables: `34` <dbl>, ...12 <lgl>, ...13 <lgl>
```

or look at the tail `tail(df1)`:

```
tail(df1, 4)
#> # A tibble: 4 x 13
#>     AHE      `25`     `26`     `27`     `28`     `29`     `30`     `31`     `32`
#>   <dbl>     <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
#> 1    65   7.27e-5  2.91e-4  1.45e-4  2.91e-4  3.63e-4  4.36e-4  5.09e-4  0.00102
#> 2    70   5.09e-4  9.45e-4  4.36e-4  1.16e-3  1.16e-3  1.31e-3  1.31e-3  0.00313
#> 3    NA  NA       NA       NA       NA       NA       NA       NA       NA
#> 4    NA  NA       NA       NA       NA       NA       NA       NA       NA
#> # ... with 4 more variables: `33` <dbl>, `34` <dbl>, ...12 <lgl>, ...13 <lgl>
```

Quick View of the Data

A few things are apparent from viewing the data in the console, that may not have been immediately obvious from the display of `View()`

The column names are strings:

```
colnames(df1)
#>  [1] "AHE"   "25"    "26"    "27"    "28"    "29"    "30"    "31"    "32"
#> [10] "33"    "34"    "...12" "...13"
```

In R, a string is called a character, it is usually displayed with quote-marks, as in “25”.

There are missing values – those are the `NA` values.

The variables described as `<dbl>` are double-precision floating-point number – as opposed to, say, integers.

Quick View of the Data

We see missing values in the last rows of the dataframe, but the output of `head()` and `tail()` is truncated before the last columns, where there are more missing values. These missing values are not apparent from viewing the spreadsheet in Excel, but they could be seen with `View(df1)`.

Viewing a selected slice of the dataframe:

```
df1[ncol(df1)]  # last column
#> # A tibble: 31 x 1
#>    ...13
#>    <lgl>
#>  1 NA
#>  2 NA
#>  3 NA
#>  4 NA
#>  5 NA
#>  6 NA
#>  7 NA
#>  8 NA
#>  9 NA
#> 10 NA
#> # ... with 21 more rows
```

where `ncol()` counts the number of columns – There is also `nrow()`.

Quick View of the Data

Viewing a selected slice of the dataframe:

```
df1[,(ncol(df1)-4):ncol(df1)]  # last 5 columns
#> # A tibble: 31 x 5
#>       `32`    `33`    `34` ...12 ...13
#>      <dbl>   <dbl>   <dbl> <lgl> <lgl>
#>  1 0.00102 0.00145 0.00102 NA    NA
#>  2 0.00174 0.00102 0.00124 NA    NA
#>  3 0.00189 0.00204 0.00182 NA    NA
#>  4 0.00174 0.00211 0.00225 NA    NA
#>  5 0.00196 0.00305 0.00204 NA    NA
#>  6 0.00414 0.00480 0.00363 NA    NA
#>  7 0.00349 0.00283 0.00334 NA    NA
#>  8 0.00669 0.00574 0.00661 NA    NA
#>  9 0.00371 0.00392 0.00392 NA    NA
#> 10 0.00574 0.00523 0.00661 NA    NA
#> # ... with 21 more rows
```

Quick View of the Data

Viewing a selected slice of the dataframe:

```
df1[6:7]  # if you know exactly which columns to select
#> # A tibble: 31 x 2
#>       `29`    `30`
#>      <dbl>   <dbl>
#>  1 0.00145 0.00153
#>  2 0.00131 0.00109
#>  3 0.00283 0.00240
#>  4 0.00138 0.00276
#>  5 0.00320 0.00240
#>  6 0.00443 0.00451
#>  7 0.00371 0.00269
#>  8 0.00705 0.00727
#>  9 0.00429 0.00443
#> 10 0.00690 0.00661
#> # ... with 21 more rows
```

Quick View of the Data

with tidyverse pipe | displays in reverse order:

```
library(tidyverse)
df1 %>% select(last_col(offset=0:4), everything()) %>% head(5)
#> # A tibble: 5 x 13
#>   ...13 ...12    `34`    `33`    `32`   AHE    `25`    `26`    `27`    `28`
#>   <lgl> <lgl>   <dbl>   <dbl>   <dbl> <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
#> 1 NA    NA    0.00102 0.00145 0.00102     5 0.00298 0.00218 0.00204 0.00211
#> 2 NA    NA    0.00124 0.00102 0.00174     6 0.00116 0.00167 0.00109 0.00109
#> 3 NA    NA    0.00182 0.00204 0.00189     7 0.00247 0.00262 0.00254 0.00167
#> 4 NA    NA    0.00225 0.00211 0.00174     8 0.00240 0.00225 0.00167 0.00240
#> 5 NA    NA    0.00204 0.00305 0.00196     9 0.00356 0.00327 0.00283 0.00291
#> # ... with 3 more variables: `29` <dbl>, `30` <dbl>, `31` <dbl>
```

The %>% symbol is a “pipe” to conveniently transforme the original `tibble` in a series of stages, convenient for debugging.

The %>% pipe is from the `tidyverse` family, not from base R.

Quick View of the Data

A more compact display of the last column:

```
library(tidyverse)
df1[ncol(df1)] %>% data.frame %>% head(5)
#>   ...13
#> 1    NA
#> 2    NA
#> 3    NA
#> 4    NA
#> 5    NA
```

Here we transform the `tibble` to a dataframe before displaying the head. A dataframe has a more compact display.

The default `dataframe` does not support integers as names and therefore prepends an X: “25” becomes “X25”. In `tidyverse`, the `tibble` is an extension of the `dataframe` which supports numbers as column names. Let’s see that now.

Read the Data as CSV

It is often convenient to work with `csv` files (`csv` stands for comma-separated values), because `csv` files have a simple structure and can be read with other statistical software.

Save the excel spreadsheet as a plain `csv` file and open it with:

```
df1.csv <- read.csv("Age_HourlyEarnings.csv",
                    skip=1,
                    header=TRUE,
                    stringsAsFactors=FALSE)
```

Most of the time, you want to make sure that strings are not converted to factors. This is done with the argument `stringsAsFactors=FALSE`, which is otherwise set to `TRUE` by default.

By contrast, `readxl` does not convert strings as factors by default.

Let's see the differences between `df1` and this `df`. We can check the structure of an object with `str()`.

Compare dataframe and tibble

This is the structure of the data imported with `read_xlsx()`:

```
str(df1)
#> tibble [31 x 13] (S3: tbl_df/tbl/data.frame)
#>  $ AHE  : num [1:31] 5 6 7 8 9 10 11 12 13 14 ...
#>  $ 25   : num [1:31] 0.00298 0.00116 0.00247 0.0024 0.00356 ...
#>  $ 26   : num [1:31] 0.00218 0.00167 0.00262 0.00225 0.00327 ...
#>  $ 27   : num [1:31] 0.00204 0.00109 0.00254 0.00167 0.00283 ...
#>  $ 28   : num [1:31] 0.00211 0.00109 0.00167 0.0024 0.00291 ...
#>  $ 29   : num [1:31] 0.00145 0.00131 0.00283 0.00138 0.0032 ...
#>  $ 30   : num [1:31] 0.00153 0.00109 0.0024 0.00276 0.0024 ...
#>  $ 31   : num [1:31] 0.001308 0.000509 0.001744 0.00218 0.002616 ...
#>  $ 32   : num [1:31] 0.00102 0.00174 0.00189 0.00174 0.00196 ...
#>  $ 33   : num [1:31] 0.00145 0.00102 0.00204 0.00211 0.00305 ...
#>  $ 34   : num [1:31] 0.00102 0.00124 0.00182 0.00225 0.00204 ...
#>  $ ...12: logi [1:31] NA NA NA NA NA NA ...
#>  $ ...13: logi [1:31] NA NA NA NA NA NA ...
```

Compare dataframe and tibble

This is the structure of the data imported with `read.csv()`:

```
str(df1.csv)
#> 'data.frame':    31 obs. of  13 variables:
#>  $ AHE: int  5 6 7 8 9 10 11 12 13 14 ...
#>  $ X25: num  0.00298 0.00116 0.00247 0.0024 0.00356 ...
#>  $ X26: num  0.00218 0.00167 0.00262 0.00225 0.00327 ...
#>  $ X27: num  0.00204 0.00109 0.00254 0.00167 0.00283 ...
#>  $ X28: num  0.00211 0.00109 0.00167 0.0024 0.00291 ...
#>  $ X29: num  0.00145 0.00131 0.00283 0.00138 0.0032 ...
#>  $ X30: num  0.00153 0.00109 0.0024 0.00276 0.0024 ...
#>  $ X31: num  0.001308 0.000509 0.001744 0.00218 0.002616 ...
#>  $ X32: num  0.00102 0.00174 0.00189 0.00174 0.00196 ...
#>  $ X33: num  0.00145 0.00102 0.00204 0.00211 0.00305 ...
#>  $ X34: num  0.00102 0.00124 0.00182 0.00225 0.00204 ...
#>  $ X  : logi  NA NA NA NA NA NA ...
#>  $ X.1: logi  NA NA NA NA NA NA ...
```

Compare dataframe and tibble

The `tibble` is a modern evolution of the `dataframe` available in base R. It is part of the family of packages developed by RStudio and Hadley Wickham that go under the name `tidyverse`. The description states "The tidyverse is an opinionated collection of R packages designed for data science."

These packages include, among others:

- `ggplot2` the most popular plotting library
- `scales` a companion to `ggplot2` that assists in scaling axes and formatting labels
- `dplyr`, `tidyr` provide various utilities to conveniently manipulate data.
- `readxl` to read excel files and convert them to a `tibble`
- `stringr` to manipulate strings
- `forcats`, `readr`, `purr`, etc.

Go to https://www.tidyverse.org/ and download the cheatsheets

Older, now deprecated packages that people still use include `plyr`, `reshape2`.

- Be careful if you use both `plyr` and `dplyr` because they both use a `summarise` function, and unexpected results can occur. Instead of `summarise`, write `dplyr::summarise` or `plyr::summarise` to make sure you are using the desired function.

Clean the Data

Check basic information:

```
nrow(df1)
ncol(df1)
colnames(df1)
is.na(df1)
complete.cases(df1)
```

Remove empty columns:

```
df1 <- Filter(function(x)!all(is.na(x)), df1)
```

Remove empty columns, including empty strings and 0 values:

```
Filter(function(x)!all(is.na(x) || is.null(x) || x == "" || x == 0), df1)
```

See if any rows have missing values:

```
complete.cases(df1)
#>  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#> [13]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#> [25]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
```

The last two rows have missing values!

Clean the Data

Remove empty rows:

```
df1 <- df1[complete.cases(df1), ]
```

Save cleaned data:

```
save(df1, file = "Age_HourlyEarnings_clean_wide.RData")
```

Or see below how to save to a `csv` or to an `xlsx` file.

Clean the Data | with dplyr

The `dplyr` does certain things more intuitively than base R. Let's clean the data again!

```
df1 <- read_xlsx("Age_HourlyEarnings.xlsx", col_names=TRUE, skip=1, trim_ws=TRUE)
```

Because there are missing values in both the rows and the columns, we have to proceed with caution. It would be tempting to do this:

```
df1 %>% filter_all(all_vars(complete.cases(.)))
#> # A tibble: 0 x 13
#> # ... with 13 variables: AHE <dbl>, 25 <dbl>, 26 <dbl>, 27 <dbl>, 28 <dbl>,
#> #   29 <dbl>, 30 <dbl>, 31 <dbl>, 32 <dbl>, 33 <dbl>, 34 <dbl>, ...12 <lgl>,
#> #   ...13 <lgl>
```

but then the whole dataset would be filtered out except the column names!

Instead, proceed in two separate steps.

Remove the empty rows:

```
df1 %>% select_if(~!(all(is.na(.)))) -> df1
```

Remove the empty columns:

```
df1 %>% filter_all(any_vars(!is.na(.))) -> df1
```

Transform the Data

The data is currently in wide format, it will be convenient to convert it to long format:

```
library("tidyr")
df2 <- gather(df1, Age, Probability, -AHE)
df2$Age <- as.integer(df2$Age)  # convert the string to an integer
```

Quick check:

```
head(df2, 6)
#> # A tibble: 6 x 3
#>     AHE Age   Probability
#>   <dbl> <chr>       <dbl>
#> 1     5 25        0.00298
#> 2     6 25        0.00116
#> 3     7 25        0.00247
#> 4     8 25        0.00240
#> 5     9 25        0.00356
#> 6    10 25        0.00516
```

Save the long-form data as `xlsx` for reference

```
library("writexl")
write_xlsx(df2, "Age_HourlyEarnings_clean_long.xlsx")
```

Wide Format

This is what data in wide format looks like:

	A	B	C	D	E	F
1	AHE	25	26	27	28	29
2	5	0.00298	0.00218	0.002035	0.002108	0.001454
3	6	0.001163	0.001672	0.00109	0.00109	0.001308
4	7	0.002471	0.002616	0.002544	0.001672	0.002835
5	8	0.002398	0.002253	0.001672	0.002398	0.001381
6	9	0.003561	0.003271	0.002835	0.002907	0.003198
7	10	0.00516	0.005233	0.003997	0.005233	0.004433
8	11	0.003779	0.004506	0.003997	0.003779	0.003707
9	12	0.007559	0.007341	0.007413	0.006977	0.00705
10	13	0.003707	0.003489	0.003634	0.003561	0.004288

Long Format

This is what data in long format looks like:

	A	B	C	D	E	F
1	AHE	Age	Probability			
2	5	25	0.00297987			
3	6	25	0.00116288			
4	7	25	0.00247111			
5	8	25	0.00239843			
6	9	25	0.00356131			
7	10	25	0.00516026			
8	11	25	0.00377934			
9	12	25	0.00755869			
10	13	25	0.00370666			
11	14	25	0.0057417			

Marginal Distribution

The general expression for the marginal distribution:

$$\Pr(Y = y) = \sum_{i=1}^{n} \Pr(X = x_i, Y = y)$$

In particular, for $Age = 25$, we have

$$\Pr(Age = 25) = \sum_{ahe=5}^{70} \Pr(AHE = ahe, Age = 25)$$

We can now calculate these conveniently with the long-form dataframe.

Marginal Distribution

Add up the probabilities by Age, using the long-form dataframe:

```
sum(df2[df2$Age == 25,]$Probability)
#> [1] 0.0849
sum(df2[df2$Age == 26,]$Probability)
#> [1] 0.0922
sum(df2[df2$Age == 27,]$Probability)
#> [1] 0.0855
sum(df2[df2$Age == 28,]$Probability)
#> [1] 0.0934
sum(df2[df2$Age == 29,]$Probability)
#> [1] 0.103
sum(df2[df2$Age == 30,]$Probability)
#> [1] 0.105
sum(df2[df2$Age == 31,]$Probability)
#> [1] 0.104
sum(df2[df2$Age == 32,]$Probability)
#> [1] 0.108
sum(df2[df2$Age == 33,]$Probability)
#> [1] 0.109
sum(df2[df2$Age == 34,]$Probability)
#> [1] 0.115
```

Marginal Distribution

To avoid copy-pasting, you can use a split/apply technique:

```
sapply(split(df2, df2$Age), function(x) sum(x$Probability))
#>     25     26     27     28     29     30     31     32     33     34
#> 0.0849 0.0922 0.0855 0.0934 0.1035 0.1047 0.1039 0.1081 0.1088 0.1150
```

And you can always write a loop:

```
for (age in 25:34){
    print(sum(df2[df2$Age == age,]$Probability))
}
#> [1] 0.0849
#> [1] 0.0922
#> [1] 0.0855
#> [1] 0.0934
#> [1] 0.103
#> [1] 0.105
#> [1] 0.104
#> [1] 0.108
#> [1] 0.109
#> [1] 0.115
```

Marginal Distribution

Using data in wide format is sometimes convenient.

Add up for all MHE rows:

```
colSums(df1[,names(df1) != "AHE"])
#>     25     26     27     28     29     30     31     32     33     34
#> 0.0849 0.0922 0.0855 0.0934 0.1035 0.1047 0.1039 0.1081 0.1088 0.1150
```

Add up for all Age columns, using the wide-form dataframe:

```
rowSums(df1[,names(df1) != "AHE"])
#>  [1] 0.01708 0.01192 0.02202 0.02115 0.02784 0.04572 0.03474 0.06985 0.03743
#> [10] 0.06301 0.04041 0.03249 0.05669 0.02842 0.05909 0.03590 0.05233 0.06766
#> [19] 0.03649 0.04150 0.04237 0.04862 0.03125 0.02144 0.01773 0.00858 0.00974
#> [28] 0.00443 0.01410
```

Note that:

```
sum(colSums(df1[,names(df1) != "AHE"]))
#> [1] 1
```

Save your results

You may want to append your results to the original data, the way you would in a spreadsheet. This may be convenient if you plan to save the data in a spreadsheet at the end of the project.

To avoid altering the clean dataframe `df1`, let's make a copy first:

```
d <- df1  # make a copy to preserve original dataframe
```

Append sums to last column:

```
d$Total <- rowSums(df1[,names(df1) != "AHE"])
```

Append sums to last row: add NA to the first column, for alignment

```
d <- rbind(d, c(NA, colSums(d[,names(d) != "AHE"])))
```

Save it as a `csv` file:

```
write.csv(d, file="Age_HourlyEarnings_clean_wide_sums.csv")
```

Save it as a `xlsx` file:

```
library("writexl")  # remember to install the package first
write_xlsx(d, "Age_HourlyEarnings_clean_wide_sums.xlsx")
```

If you haven't set your working directory, run `getwd()` to see where the files were saved.

Compute Weighted Means by Age: Base R

Using base R | See below for other methods

Here is an approach based on split/apply:

```
sapply(split(df2, df2$Age), function(x) weighted.mean(x$AHE, x$Probability))
#>   25   26   27   28   29   30   31   32   33   34
#> 17.6 19.0 19.7 20.2 21.2 21.8 22.6 23.7 23.3 24.1
```

There are many ways to achieve the same result. The `tidyverse` offers more intuitive functions for this purpose. We explore them next.

Compute Weighted Means by Age: plyr

The `dplyr` package is a crowd's favorite:

```
detach("package:plyr")  # to avoid conflict with dplyr
library("dplyr")
df2 %>%
    group_by(Age) %>%
    summarise(AHE = weighted.mean(AHE, Probability)) %>%
    head(5)
#> # A tibble: 5 x 2
#>     Age   AHE
#>   <int> <dbl>
#> 1    25  17.6
#> 2    26  19.0
#> 3    27  19.7
#> 4    28  20.2
#> 5    29  21.2
```

One advantage of piping is that you can write the variable name directly — that is, `Age` instead of `df2$Age`

Compute Weighted Means by Age: data.table

Another popular choice is based on the `data.table` library. The `data.table` library is the most efficient for large datasets.

```
library("data.table")
head(setDT(df2)[, .(AHE = weighted.mean(AHE, Probability)), Age])
#>    Age  AHE
#> 1:  25 17.6
#> 2:  26 19.0
#> 3:  27 19.7
#> 4:  28 20.2
#> 5:  29 21.2
#> 6:  30 21.8
```

Law of Iterated Expectations

In general,

$$E[Y] = \sum_{i=1}^{n} E[Y|X = x_i] \Pr(X = x_i)$$

In particular,

$$E[AHE] = \sum_{age=25}^{34} E[AHE|Age = age] \Pr(Age = age)$$

Compute Variance by Age

Similar to what we did with the function `weighted.mean`, but here we roll our own `weighted.variance`:

```
weighted.variance <- function(x,w) sum(w*(x-weighted.mean(x,w))^2)/sum(w)
dv <- sapply(split(df2, df2$Age), function(x) weighted.variance(x$AHE, x$Probability))
head(dv)
#>    25    26    27    28    29    30
#>  95.9 129.0 128.8 142.7 152.2 162.8
```

As information about the sample size is not available, we do not attempt to remove any bias that may be present in the formula.

Compute Variance for all Ages

To compute the variance for all ages, we first save the mean computed earlier and merge with the variance computed above.

```
df2 %>% group_by(Age) %>%
    dplyr::summarise(Probability = sum(Probability)) -> df

df2 %>% group_by(Age) %>%
    dplyr::summarise(Mean = weighted.mean(AHE, Probability)) %>%
    left_join(df, by = "Age") -> df

df2 %>% group_by(Age) %>%
    dplyr::summarise(Var = weighted.variance(AHE, Probability)) %>%
    left_join(df, by = "Age") -> df

head(df, 3)
#> # A tibble: 3 x 4
#>     Age   Var  Mean Probability
#>   <int> <dbl> <dbl>       <dbl>
#> 1    25  95.9  17.6      0.0849
#> 2    26 129.   19.0      0.0922
#> 3    27 129.   19.7      0.0855
```

Compute Variance for all Ages

We can now compute the weighted mean and weighted variance:

```
df %>% summarize(Mean = weighted.mean(Mean, Probability)/sum(Probability))
#>   Mean
#> 1 21.5

df %>% summarize(Var = weighted.mean(Var, Probability)/sum(Probability))
#>   Var
#> 1 161
```

The expected value of the square could be computed likewise

```
weighted.squared.exp <- function(x,w) sum(w*x^2)/sum(w)
```

Visualize the Data

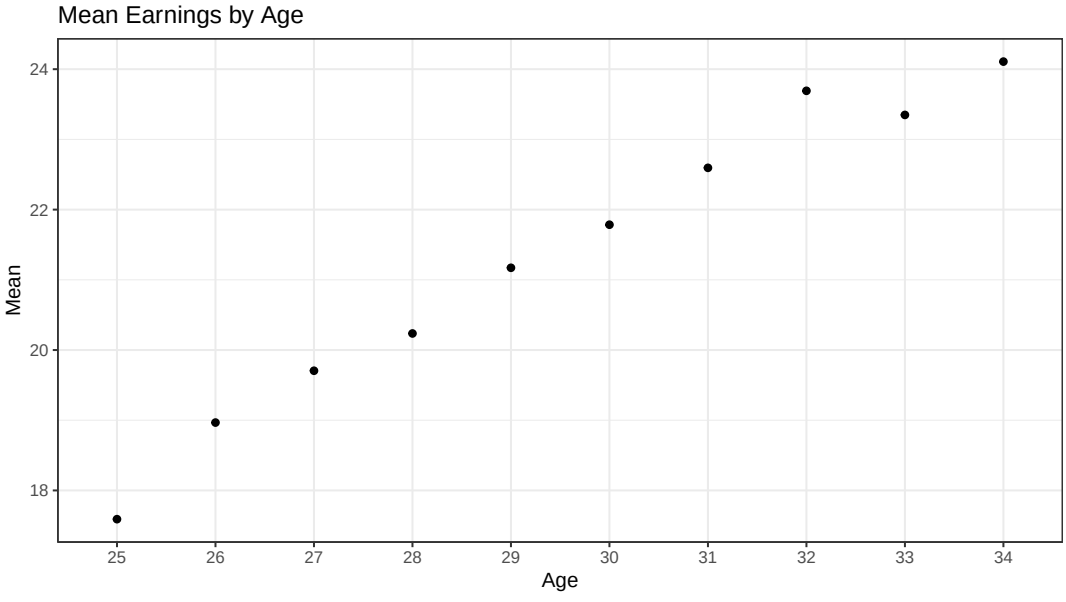
Mean Earnings by Age:

```
library("ggplot2")
ggplot(data=dm, aes(x=Age, y=Mean)) +
    geom_point() +
    ggtitle("Mean Earnings by Age") +
    theme_bw()
```

Variance of Earnings by Age:

```
ggplot(data=dv, aes(x=Age, y=Variance)) +
    geom_point() +
    ggtitle("Variance of Earnings by Age") +
    theme_bw()
```

Visualize the Data



Visualize the Data